Administración de variabilidad en una línea de productos basada en modelos

KELLY GARCÉS PERNETT

UNIVERSIDAD DE LOS ANDES

FACULTAD DE INGENIERÍA

DEPARTAMENTO DE INGENIERÍA DE SISTEMAS

MAESTRÍA EN INGENIERÍA DE SISTEMAS Y COMPUTACIÓN

BOGOTÁ D.C., 2007

Administración de variabilidad en una línea de productos basada en modelos

KELLY GARCÉS PERNETT

Tesis de grado

Directora

Phd. Rubby Casallas Gutiérrez

Profesora asociada del departamento de sistemas y computación

UNIVERSIDAD DE LOS ANDES

FACULTAD DE INGENIERÍA

DEPARTAMENTO DE INGENIERÍA DE SISTEMAS

MAESTRÍA EN INGENIERÍA DE SISTEMAS Y COMPUTACIÓN

BOGOTÁ D.C. 2007

Nota de aceptación

_____

_____

_____

_____

Director de proyecto

_____

Jurado

_____

Jurado

_____

Bogotá D.C., Enero de 2007

Tabla de contenido

Tabla de figuras

1. Introducción

El ingeniero de software siempre se ha preocupado por mejorar la productividad en el proceso de desarrollo. Para alcanzar este objetivo, uno de los derroteros ha sido la reutilización de software. La reutilización de software es el proceso de crear sistemas a partir de software existente en lugar de construirlos desde cero. Además de aumentar la productividad, la reutilización reduce los riesgos, los costos del desarrollo y facilita el mantenimiento del software. Alcanzar dichos beneficios ha sido motivación de varias estrategias, entre ellas las composicionales y las generativas [27]. Las estrategias composicionales construyen sistemas a partir de componentes de bajo nivel que se encuentran en un depósito. Las estrategias generativas reutilizan un proceso de generación, son específicas a ciertos dominios y adoptan arquitecturas de referencia.

Al comparar estas estrategias se encuentran ventajas y desventajas. Las estrategias generativas son aplicables a un cierto dominio mientras que las composicionales son aplicables a una amplia variedad de aplicaciones. Adicionalmente, los componentes son más modulares, se auto contienen y son mantenibles. Su desventaja es que rara vez son perfectos o lo suficientemente genéricos. En las estrategias composicionales los componentes son dependientes de la tecnología de implementación y en las generativas, al usar artefactos de bajo nivel, se mezclan preocupaciones que dificultan el mantenimiento.

La rápida evolución tecnológica y el cambio en los requerimientos de negocio exigen nuevas estrategias que solucionen los inconvenientes descubiertos en las estrategias mencionadas. Un paradigma de reutilización que pretende encarar dichos desafíos son las Líneas de Productos Software (Software Product Lines SPL). Una SPL se define como un conjunto de sistemas software que satisfacen necesidades específicas de un segmento de mercado [28]. Los productos de una SPL comparten características en común y son desarrollados a partir de componentes reutilizables. Los componentes reutilizables, llamados activos de la línea, son entrada a los procesos definidos para la SPL. Los procesos clásicos de una SPL son: el desarrollo de activos, el desarrollo de productos y la administración del proceso de desarrollo.

En una SPL es necesario explotar lo común y manejar la variabilidad. Manejar la variabilidad significa soportar la producción de productos diferentes, que estén dentro del alcance de la SPL, a partir de los activos de la misma. Como herramienta para describir la variabilidad entre productos el SEI propuso los modelos de rasgos. Un modelo de rasgos captura las características comunes y variables, existentes entre los productos de una SPL, que son relevantes a los

usuarios. De esta forma es imaginable que a partir de una selección de rasgos realizada por el usuario se pueda construir un producto con los activos apropiados.

Por otro lado, la arquitectura basada en modelos (Model Driven Architecture MDA) es una propuesta del Object Management Group (OMG), que plantea la construcción de software a partir de modelos a distintos niveles de abstracción para guiar todo el proceso de desarrollo. Los modelos generalmente se relacionan con los diagramas de clase, de colaboración o de estados UML, estos modelos son usados como documentación, sin embargo teniendo las herramientas adecuadas, un conjunto de modelos puede ser usado para generar toda una aplicación o parte de ella. El proceso de desarrollo MDA propone transformaciones sucesivas entre diferentes tipos de modelos para llegar al código fuente de forma sistemática. MDA propone la construcción de modelos independientes de la plataforma (PIM) y de modelos dependientes de la plataforma (PDM) a partir de los cuales se pueda producir un modelo específico de la plataforma (PSM). Gracias a esta separación se puede partir de un mismo PIM y conseguir aplicaciones implementadas sobre diferentes plataformas (J2EE, .Net, Corba).

De la unión de SPL y MDA surgen las Líneas de Producto Software basadas en modelos (MD-SPL) donde los activos son modelos, metamodelos y transformaciones. Los modelos son elementos de primera clase en el desarrollo de aplicaciones, más que simples elementos de representación y comunicación. Un metamodelo abstrae los conceptos de un dominio (negocio, tecnología, plataforma tecnológica). Un modelo utiliza los conceptos de un metamodelo para describir un sistema en un dominio particular. En este sentido decimos que un modelo es conforme a un metamodelo. Los modelos son transformados en otros usando transformaciones. Las transformaciones generan modelos destino a partir de modelos origen.

El propósito principal de MDA no es la creación de familias de productos. Sin embargo, la separación de dominios y la naturaleza generativa, hacen de MDA un enfoque de gran utilidad para la creación de SPLs. Unir SPL y MDA permite aprovechar las ventajas de los dos enfoques. Una SPL permite la reutilización de los artefactos y de los procesos con lo cual se gana productividad. Por otra parte MDA pretende que los desarrolladores se concentren en desarrollar los modelos y las transformaciones, ya que el código podría generarse automáticamente a través de estos.

En este trabajo se presenta una propuesta para manejar variabilidad en MD-SPL. En esta propuesta se separan conceptos relacionados con SPLs en diferentes dominios, y se construyen como activos de la línea modelos de rasgos, metamodelos y reglas de transformación. Los metamodelos soportan la variabilidad de un dominio específico, es decir, que a partir de ellos se puede construir un amplio conjunto de modelos variables. Los modelos de rasgos describen la variabilidad de los modelos destino. Las reglas transforman los

modelos de un dominio origen en diferentes modelos de un dominio destino, de acuerdo con una selección de rasgos del dominio destino. Las reglas de transformación son de tres tipos: (1) base, (2) de control y (3) específicas. Las primeras establecen una correspondencia para transformar elementos conformes al metamodelo origen en otros elementos conformes al metamodelo destino. Estas reglas son utilizadas para generar las características comunes a todos los modelos destino posibles. Las segundas identifican elementos del modelo origen que deben ser transformados de una manera particular, crean elementos en el modelo destino e invocan reglas específicas atendiendo a una selección de rasgos. Las reglas específicas configuran los elementos del modelo destino que se han creado de acuerdo a los parámetros colocados por las reglas de control. Las reglas tipo 2 y 3 son utilizadas para generar las características variables de un modelo destino. La generación sucesiva de modelos destino diferentes permite crear aplicaciones con características variables en cada dominio. Utilizando metamodelos se logra ampliar el alcance de la SPL y separar los dominios. Adicionalmente, utilizando modelos de rasgos y reglas de transformación asociadas a éstos se logra crear aplicaciones variables.

La propuesta planteada se materializa en un plug-in para eclipse llamado Qualdev-MDSPL, el cual facilita la generación de aplicaciones variables a través de la creación de modelos origen, selección de rasgos en cada dominio y la generación automática a partir de transformaciones.

El problema y la propuesta son ilustrados con Cupi2, un proyecto del grupo de construcción de software de la Universidad de los Andes, en el cuál se describe un plan de estudios para la enseñanza/aprendizaje de la programación de computadores.

El documento se encuentra organizado de la siguiente manera: En el capítulo 2 se presenta el planteamiento del problema y los objetivos de la investigación. El capítulo 3 presenta el estado del arte del paradigma SPL, del enfoque MDE/MDA y de MD-SPL. El capítulo 4 describe la solución propuesta, el capítulo 5 el plug-in que implementa la solución propuesta, el capítulo 6 una experimentación de la solución propuesta. Finalmente los capítulos 7, 8 y 9 presentan aportes de la investigación, trabajos futuros y conclusiones respectivamente.

2. Planteamiento del problema y objetivos

Este trabajo de investigación está contextualizado dentro de un marco de trabajo general cuyo propósito es clarificar el proceso de construcción de líneas de producto basadas en modelos. En [2] se propone un enfoque de MD-SPL donde los activos son modelos, metamodelos y transformaciones. Los metamodelos abstraen las diversas preocupaciones del dominio del problema; metamodelos de lógica de negocio, metamodelos de arquitectura y metamodelos de plataforma tecnológica, son algunos de los metamodelos sugeridos. Las transformaciones se definen de forma lineal: la primera transformación se construye para transformar modelos conformes al metamodelo de la lógica de negocio, en modelos conformes al metamodelo de la arquitectura de desarrollo. La segunda transformación se construye para transformar modelos conformes al metamodelo de arquitectura de desarrollo, en modelos conformes al metamodelo de la plataforma tecnológica. Durante el desarrollo de un producto se crean modelos conforme al metamodelo de lógica de negocio, se ejecutan las transformaciones hasta llegar al modelo de plataforma tecnológica y éste es transformado a texto (código fuente).

Una aplicación concreta del enfoque descrito es presentada en [25]. La experimentación de dicha propuesta es hecha en el dominio de Cupi2. Cupi2 [7] es un proyecto del grupo de construcción de software de la Universidad de Los Andes que tiene como objetivo la búsqueda de nuevas maneras para enfrentar el problema de enseñar a programa. Una de las etapas de este proyecto considera la reestructuración de los cursos del área de programación del currículo de pregrado, la cual consiste en alinear los cursos con siete ejes temáticos, llamados ejes de la programación. Cada eje delimita conocimientos y habilidades que el estudiante debe desarrollar. Un curso está dividido por niveles. Cada nivel está acompañado de objetivos pedagógicos, ejercicios y ejemplos. Los objetivos pedagógicos apuntan a desarrollar conocimientos y habilidades en uno o varios de los ejes temáticos. Un ejercicio es un proyecto completo de programación, en el cual hay partes que el estudiante debe completar y partes que están dadas. Un ejemplo es una aplicación funcional que ilustra algunos de los conocimientos y habilidades que se buscan generar con el ejercicio [30].

En [26] se presentan dos líneas de producto que hacen parte del proyecto Cupi2. Ambas generan herramientas de apoyo al proceso educativo, PLACENTA crea entrenadores y LED crea aplicaciones para animar estructuras de datos. Los activos de las SPL son componentes flexibles que se conectan a una plataforma.

Por su parte [25] propone una línea de productos basada en modelos que construye parte de los ejemplos para algunos niveles de Cupi2. Algunas

características de los ejemplos que hacen de ellos un dominio potencial para MD-SPL son:

- Son incrementales, el desarrollo de un ejemplo puede requerir conocimientos y habilidades adquiridas en ejemplos previos.
- Están alineados por ejes temáticos.
- Comparten similitudes y diferencias, son variables.
- Su desarrollo compromete muchos recursos.
- Ajustarlos para satisfacer los objetivos pedagógicos es una operación manual propensa a errores.
- Masificación sin perder calidad y uniformidad, Cupi2 tiene una población objetivo de estudiantes con tendencia al aumento, lo cual implicaría el desarrollo de un mayor número de ejemplos. Es deseable que las aplicaciones desarrolladas mantengan calidad y estén alineadas por los ejes temáticos.

Todos los ejemplos tienen en común que:

- Son aplicaciones monousuario sin requerimientos no funcionales complejos.
- Son escritos en java.
- Se estructuran por tres componentes: el mundo, la interfaz y las pruebas. El componente mundo implementa los conceptos del negocio, sus atributos y relaciones. El componente interfaz usuario implementa la visualización de la información y la interacción entre el usuario y componente del mundo. El componente de las pruebas implementa pruebas unitarias de la funcionalidad ofrecida por el componente mundo.

Para soportar las características comunes y diferentes de la MD-SPL Cupi2 se han construido tres metamodelos: mundo, arquitectura y tecnología. Cada metamodelo especifica un dominio diferente, la separación de preocupaciones se logra independizando los conceptos del mundo del problema de los conceptos de la arquitectura y la plataforma tecnológica.

El metamodelo de mundo o metamodelo de lógica de negocio describe los conceptos esenciales del mundo del problema en los ejemplos Cupi2. El metamodelo de arquitectura, muestra los conceptos de interfaz de usuario y los servicios ofrecidos y el metamodelo de la plataforma tecnológica, en este caso Java, muestra los conceptos propios del lenguaje como por ejemplo paquetes, clases, métodos y atributos.

Las transformaciones permiten enriquecer cada modelo origen (mundo) con los conceptos de los modelos destino (arquitectura y tecnología). Las transformaciones entre los modelos conformes a metamodelos citados son

lineales: primero se aplica una transformación del metamodelo del mundo al metamodelo de arquitectura. Así, ésta transformación genera dos modelos: un modelo de interfaz y un modelo de negocio. Luego se aplica a los modelos generados, una transformación del metamodelo de arquitectura al metamodelo de Java. Por último se realiza la generación del código fuente utilizando plantillas. La figura 1 muestra el esquema de generación de un producto en una MD-SPL Cupi2.

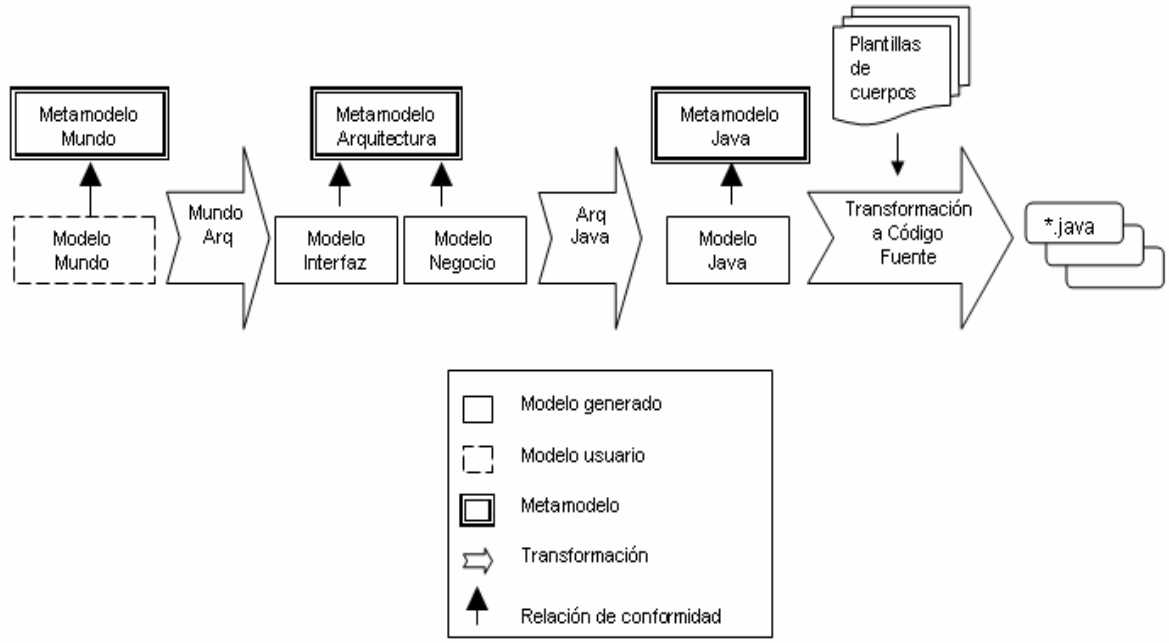

Figura 1. Esquema de generación de productos en MD-SPL Cupi2.

Uno de los compromisos principales en una SPL es el manejo de las características comunes y diferentes para generar productos que se ajusten a las preferencias del usuario, pero ¿cómo se pueden generar productos diferentes en una MD-SPL? Se muestra la problemática de generar productos diferentes usando una MD-SPL Cupi2.

Cuando se desarrolla un producto en una MD-SPL Cupi2, el usuario debe proveer un modelo de mundo, el cual es conforme a un metamodelo de mundo. Ser conformes significa que los modelos están delimitados por los conceptos, las relaciones y la semántica de su metamodelo. Cuando un modelo es creado podemos variar el número de elementos y la forma cómo estos se relacionan. La figura 2 muestra dos modelos de mundo, uno para el ejemplo discotienda y otro para el ejemplo exposición de vehículos. El ejemplo discotienda es una aplicación que maneja la información de una tienda virtual de canciones en formato MP3. El ejemplo exposición de vehículos es una aplicación que maneja la información de una exposición de vehículos [7]. En el modelo discotienda se aprecian tres

elementos (discotienda, disco y canción), mientras que en exposición de vehículos se aprecian sólo dos (exposición vehículo y vehículo). Adicionalmente la forma como se relacionan los elementos varía: en el modelo discotienda, Discotienda, conforme al concepto Agrupador, se relaciona con Disco, conforme también al concepto Agrupador; en el modelo exposición vehículos, Exposición vehículos, conforme al concepto Agrupador, está relacionado con Vehículo, conforme al concepto Simple.

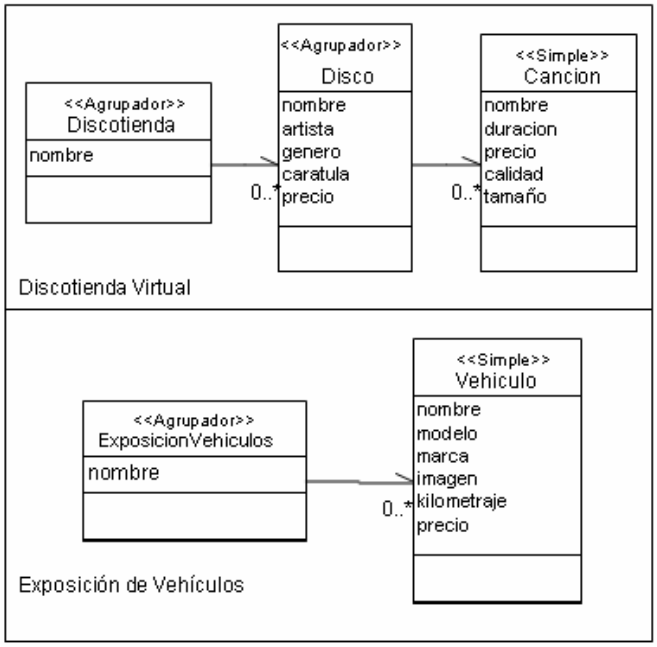

Figura 2. Modelos conformes al metamodelo de mundo.

La variabilidad que se ha descrito, a nivel de los modelos, se debe a la naturaleza generativa de MDA.

El modelo de mundo proveído por el usuario, es el modelo origen de una transformación que se encarga de generar dos modelos destino de arquitectura (modelo de interfaz y modelo de negocio). Ésta transformación se constituye de un conjunto de reglas de transformación. Una regla de transformación se define en términos de los conceptos de los metamodelos origen y destino. Las reglas de transformación implementadas en la MD-SPL Cupi2 son reglas declarativas, es decir, constan de un patrón origen y un patrón destino. El patrón origen se define en términos de uno o más elementos del metamodelo origen, mientras que el patrón destino se define en términos de uno o más elementos del metamodelo destino. El patrón origen consulta el modelo origen, mientras que la patrón destino modifica el modelo destino. En la figura 3 se distinguen los patrones de la regla de transformación simple2VistaInformacion, el patrón origen consulta todos los elementos del modelo origen que son conformes al concepto Simple del

metamodelo del mundo, los elementos que lo sean serán transformados en un elemento de tipo Vista de información en el modelo destino, tal y como lo define el patrón destino.

```
simple2VistaInformacion(origen[Simple],
                        destino[VistaInformacion])
```

Figura 3. Regla de transformación

Si se ejecuta esta regla de transformación, utilizando el modelo de la Discotienda, se encuentra la correspondencia entre Canción y elemento Simple, entonces se crea una Vista de información. Si se utiliza el modelo de la exposición de vehículos como origen, se genera la misma Vista de información. La diferencia es que en el primer caso la Vista de información está asociada a Canción y en el segundo caso está asociada a Vehículo. El mismo resultado se obtiene al aplicar cualquier modelo en el que se encuentren elementos conformes al concepto Simple, haciendo que todas las interfazs luzcan iguales.

Así, si una transformación se ejecuta muchas veces usando diferentes modelos origen, los modelos destino tendrán características similares. Sin embargo es posible que el usuario quiera obtener modelos destino con características diferentes. Por ejemplo, es posible que en el caso de la Discotienda, el usuario no desee ver la información de cada canción, sino que prefiera visualizar su nombre agrupado en una vista.

Obtener modelos destino que se ajusten a las preferencias del usuario empleando reglas de transformación declarativas, implica modificarlas cada vez que se quiera generar un modelo destino diferente para el modelo origen. Para lograr la variabilidad en los modelos destino, se necesita además de las reglas declarativas otro tipo de reglas que generen lo variable. Aplicar o no estas reglas durante la ejecución de la transformación depende de las preferencias del usuario.

La problemática descrita revela la necesidad de manejar la variabilidad a nivel de las transformaciones. La estrategia presentada para dar solución al problema tiene los siguientes objetivos:

1. Abstraer la variabilidad de los modelos destino de una transformación en un modelo.
2. Expresar variabilidad sobre el modelo origen de una transformación de forma que al ejecutarla, se apliquen las reglas de transformación adecuadas para generar un modelo destino ajustado a las preferencias del usuario.
3. Diseñar e implementar un plug-in que facilite la generación de las aplicaciones variables a partir de modelos, metamodelos y transformaciones.

3. Estado del arte

A continuación se presenta una reseña de los principales enfoques que fueron considerados durante este proyecto. Inicialmente se presenta una reseña sobre líneas de producto software. Luego se considera el manejo de variabilidad en el contexto de líneas de producto, seguidamente se referencia arquitectura dirigida por modelos y al final líneas de producto software basada en modelos.

3.1. Líneas de producto software (SPL)

SPL es un enfoque propuesto por el SEI que promete mejorar la eficiencia y productividad en el proceso de desarrollo. Una SPL se define como un conjunto de sistemas software que satisfacen necesidades específicas de un segmento de mercado [2]. Los productos de una línea comparten características comunes y son desarrollados a partir de componentes reutilizables. Los componentes reutilizables, llamados activos de la línea, son entrada a los procesos definidos para la SPL y son usados en la producción de más de un producto. Los activos pueden ser fragmentos de código, modelos de procesos, planes, documentos o cualquier otro elemento útil en la construcción de un sistema.

Uno de los activos más importantes en una línea de producto es la arquitectura de software de la línea. Esta debe satisfacer los requerimientos del conjunto de productos y proveer un contexto en el cual otros activos como plantillas de código y artefactos de pruebas puedan ser desarrollados con la flexibilidad necesaria para satisfacer las necesidades de los distintos productos de la línea. Otro activo importante es el alcance de la línea, este delimita al conjunto de productos que los activos base son capaces de producir. El alcance define las características comunes y diferentes. De esta manera surgen las familias de productos. Como se menciona en [8], una familia de producto hace referencia al grupo de productos de software que pueden ser construidos a partir de un conjunto común de activos. Los productos en una familia por lo general comparten elementos de diseño, componentes y normas para la integración del sistema. El tamaño de la familia depende de la capacidad de la línea para unificar los activos en un sistema funcional en el que se administran conceptos relativos a reglas de negocio, arquitectura y plataforma de desarrollo.

Una SPL distingue tres procesos fundamentales: 1) Desarrollo de activos, 2) Desarrollo de productos y 3) Administración del proceso de desarrollo. El desarrollo de activos consiste en analizar la SPL y desarrollar artefactos reutilizables basados en el resultado de dicho análisis. El desarrollo de productos incluye el análisis de requerimientos, configuración, adaptación de los activos y generación del código del producto. Durante el desarrollo de activos, el ingeniero

de software debe explotar lo común y lo variable entre ellos, para generar productos diferentes [17]. La administración del proceso de desarrollo incluye todos los procesos normales de administración.

La mayoría de las propuestas de SPL coinciden en que el manejo de variabilidad es necesario dentro de las SPL. La siguiente sección resume los conceptos y algunos de los principales trabajos realizados en esta área.

3.1.1. Manejo de variabilidad en SPL

Entre los aportes para expresar variabilidad se encuentra FODA [18]. Este método fue introducido en 1990 por la Universidad Camigie Mellon (CMU) y el SEI, su metodología está relacionada con el análisis de requerimientos y el diseño de alto nivel.

El principal aporte de FODA es el modelo de rasgos. Un rasgo es una propiedad del sistema relevante para algún stakeholder (usuarios finales, expertos del dominio, desarrolladores), este puede referirse a un requerimiento, a un componente de la arquitectura o a una pieza de código [18].

Un modelo de rasgos puede ser usado para:

- Capturar los resultados del análisis de dominio.
- Manejar lo común y lo variable entre los productos de una SPL.
- Facilitar el dimensionamiento de la SPL.
- Proveer una base para configurar automáticamente los productos concretos.

Durante el análisis, los rasgos son categorizados como: obligatorios, opcionales o alternativos. Los rasgos obligatorios determinan lo común entre los productos de la línea y los rasgos opcionales y alternativos determinan el grado de variabilidad de la misma [16]:

Rasgos obligatorios o comunes: son los rasgos que deben ser proveídos por cada miembro de la SPL.

Rasgos opcionales: son aquellos rasgos proveídos por sólo algunos miembros de la SPL.

Rasgos alternativos exclusivos: dos o más rasgos son alternativos exclusivos, cuando uno y sólo uno de ellos puede ser proveído por un miembro de la SPL.

Los rasgos son organizados en diagramas de rasgos. Un modelo de rasgos puede contener uno o más diagramas de rasgos más información adicional como

descripción, restricciones globales, prioridades, entre otros. [8]. Un diagrama de rasgos es un árbol cuyos nodos representan rasgos. En el árbol se va desde los rasgos de mayor nivel de abstracción hacia los de menor nivel. Los nodos pueden ser de tres tipos: nodo raíz, nodo conjunto o nodo solitario. El nodo raíz es el nodo principal del diagrama, siempre está presente. Un nodo conjunto agrupa otros nodos y un nodo solitario es aquel que por definición no está agrupado. Los nodos solitarios pueden ser obligatorios u opcionales. Un nodo solitario es obligatorio cuando representa un rasgo relevante a todos los productos, mientras que un nodo solitario es opcional cuando representa un rasgo relevante a un producto por lo menos. Los nodos solitarios obligatorios están decorados con círculos rellenos y los opcionales con círculos vacíos. Un nodo conjunto puede agrupar nodos obligatorios y opcionales ó nodos alternativos. Cuando los nodos son alternativos, sólo uno de ellos puede ser seleccionado. Los nodos obligatorios agrupados se decoran con círculos rellenos y los opcionales con círculos vacíos. Los nodos alternativos se decoran con un arco. Un nodo conjunto puede ser también un nodo solitario.

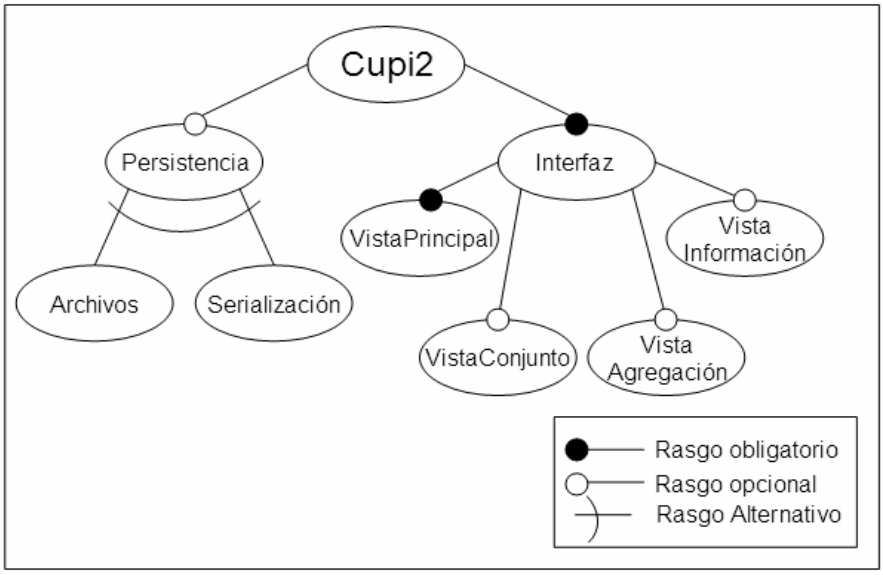

Figura 4. Diagrama de rasgos jerárquico.

La figura 4 presenta un diagrama de rasgos para la arquitectura de Cupi2. Cupi2 es el nodo raíz. Persistencia es un nodo solitario opcional y a la vez nodo conjunto, como nodo conjunto agrupa los nodos alternativos Archivos y Serialización. Interfaz es un nodo solitario obligatorio y a la vez un nodo conjunto, como nodo conjunto agrupa en nodo obligatorio Vista Principal y los nodos opcionales Vista Información, Vista Agregación y Vista Conjunto.

FODA define su proceso en tres fases:

Análisis de contexto: define el alcance de la SPL.

Modelaje del dominio: desarrolla un modelo de rasgos que es complementado por un modelo de casos de uso y un modelo de objetos.

Modelaje de arquitectura: establece la estructura del software. Los modelos arquitecturales generados son usados para construir la aplicación y están estrechamente relacionados con los modelos del dominio.

FODA no propone una representación formal del modelo de rasgos; modelos de rasgos grandes o con relaciones de composición entre los rasgos (A requiere de B) dificultan el entendimiento del diagrama jerarquizado. Tampoco presenta una aplicación concreta de los modelos de rasgos en SPL.

Para responder a las limitaciones de FODA varias propuestas han sido desarrolladas, seguidamente se presentan algunas de ellas.

3.1.1.1. Representación de modelo de rasgos

Las investigaciones realizadas por Krzysztof Czanercki y sus colaboradores [9][10] responden varias limitaciones de FODA. Los aportes están orientados a la formalización del modelo de rasgos incluyendo configuración, así:

Inclusión de cardinalidad al modelo de rasgos. La cardinalidad permite especificar la frecuencia con que un rasgo puede estar pesente dentro de una configuración. El modelo de rasgos de la figura 4 con cardinalidad se aprecia en la figura 5. La cardinalidad del nodo Persistencia [0..1] quiere decir la persistencia puede manejarse o no, la cardinalidad del nodo Vista Conjunto [0..2] quiere decir que la Vista Conjunto puede estar presente en una interfaz cero, una o dos veces, la cardinalidad del nodo Vista Agregación [1,3] quiere decir que la interfaz puede contener 1 ó 3 vistas de agregación.

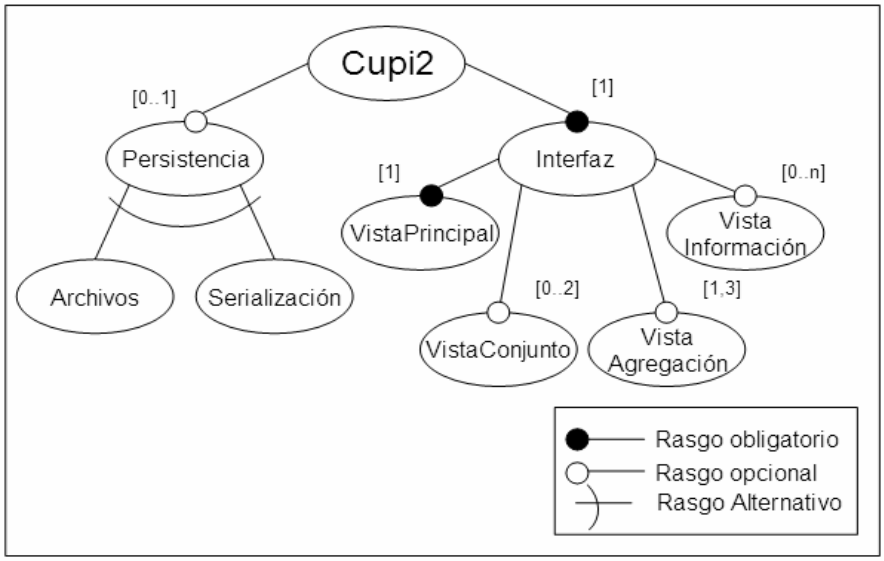

Figura 5. Diagrama de rasgos con cardinalidad.

Uso de gramáticas libres de contexto (Context Free Grammar CFG). El modelo de rasgos se representa a través de una CFG. Una gramática libre de contexto expresa un lenguaje de forma textual a partir de una colección de producciones. El diagrama de rasgos de la figura 4 se puede representar mediante 3 producciones así:

1. Cupi2:: (Persistencia)? . Interfaz
2. Persistencia:: Archivos | Serializacion
3. Interfaz:: VistaPrincipal .(VistaConjunto)? . (VistaAgregacion)? . (VistaInformacion)?

Las posibles configuraciones que se puedan escoger en un modelo de rasgos son capturadas por la CFG. Existen lenguajes como Prolog que evalúan las configuraciones y chequean el cumplimiento de restricciones.

Configuración por estados. Un modelo de rasgos describe el espacio de configuraciones de una SPL. Una configuración consiste en seleccionar los rasgos deseados del modelo de rasgos con las restricciones de variabilidad definidas por el modelo. Czarnecki resalta la necesidad de configurar por estados, pues la elección en un dominio (funcionales, arquitecturales o de plataforma tecnológica) puede reducir las opciones de configuración en los demás. Lo que propone es modularizar el modelo de rasgos, configurar por estados (módulos) y componer los modelos de rasgos.

3.1.1.2. Aplicaciones de FODA

FORM

Una extensión de FODA conocida como FORM (Feature-Oriented Reuse Method) [19] realiza el modelaje de arquitectura usando componentes orientados por objetos. Los componentes se separan por rasgos obligatorios a los productos de la SPL, también hay componentes específicos del producto.

FORM clasifica los rasgos así: capacidad, ambiente de sistema, tecnología del dominio y tecnología de implementación:

Rasgos de capacidad: caracterizan los distintos servicios, operaciones y aspectos no funcionales del sistema que son visibles al usuario final.

Rasgos del ambiente del sistema: definen el contexto externo del sistema: el ambiente de ejecución, los protocolos y las interfazs con otros sistemas.

R*asgos de la tecnología del dominio:* estos rasgos contienen las decisiones de diseño e implementación que pueden ser usadas para implementar los rasgos de capacidad. Patrones de diseño, protocolos de comunicación, estilos de arquitectura, métodos de sincronización, diseño de interfaz, son algunos ejemplos.

Rasgos de la tecnología de implementación: estos rasgos son usados por los arquitectos de software para caracterizar los detalles de implementación del sistema en una plataforma determinada.

El aporte de FORM es la categorización de los rasgos no relacionados con el usuario final.

GenVoca y AHEAD

En [3][4] se propone GenVoca, una metodología que muestra como el código de un programa individual puede ser representado por una ecuación (ver figura 6). GenVoca está inspirado en el paradigma Step-wise refinement, el cual plantea el desarrollo de programas complejos desde un programa base adicionando rasgos incrementalmente. Los rasgos son provistos por módulos llamados refinamientos. Los refinamientos encapsulan fragmentos de clases y métodos.

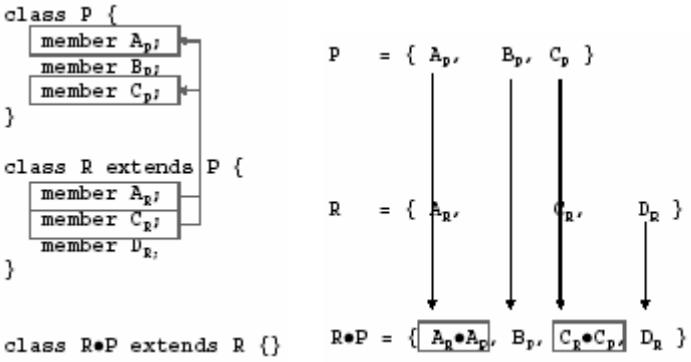

Figura 6. A la izquierda se muestra el código de un programa y a la derecha la ecuación algebraica equivalente.

Los programas base son constantes y los refinamientos son funciones que adicionan rasgos a los programas. Tanto los programas como los refinamientos en Genvoca, son colecciones de clases. Un refinamiento adiciona atributos, métodos, extiende o sobreescribe métodos. Los refinamientos se pueden implementar a través de plantillas, generadores, transformaciones u objetos.

En [5] se extiende GenVoca para expresar múltiples programas a través de un conjunto de ecuaciones, diferentes ecuaciones definen una SPL. La extensión es llamada AHEAD (Algebraic Hierarchical Equations for Application Design). AHEAD provee una herramienta llamada composer, ésta toma una ecuación como entrada, expande la ecuación recursivamente hasta llegar a expresiones simples, crea un árbol e invoca a las herramientas de composición específicas para que sinteticen las expresiones. Las expresiones simples son rasgos, los cuales tienen asociados refinamientos (fragmentos de código modulares).

Aspectos y SPL

En [1] se presenta una propuesta que usa programación orientada por aspectos para implementar la variabilidad en una SPL.
La propuesta define los siguientes pasos:

Modelo de rasgos. Se definen los rasgos de la SPL, primero se identifican los rasgos comunes a todos los productos y luego los rasgos variables.

Diseño de arquitectura. Se diseñan arquitecturas flexibles aplicando patrones.

Diseño de variabilidad usando aspectos. Se identifican los puntos de la aplicación que serán interceptados por los aspectos, se diseñan los aspectos correspondientes a los rasgos variables y clases auxiliares.

Expresión de variabilidad. Se componen los aspectos y las clases de negocio.

Generación del producto. Se genera el código y se empaqueta la aplicación.

La mayoría de las propuestas mencionadas expresan la variabilidad sobre artefactos de bajo nivel (clases, componentes), esta propuesta abstrae el significado de la variación en un modelo de rasgos que se componga con otros modelos. Aumentar el nivel de abstracción a través de modelos es uno de los objetivos de MDE, a continuación se tratan algunos de principios.

3.2. Arquitectura dirigida por modelos (MDA)

MDE (Model Driven Engineering) es un marco conceptual que plantea la construcción de software a partir de modelos, a distintos niveles de abstracción, para guiar todo el proceso de desarrollo [29]. Utilizar los modelos como entidades de primera clase ofrece mayor flexibilidad en los siguientes campos:

- Implementación: La implementación de nuevas tecnologías puede ser realizada bajo los diseños existentes.
- Integración: Tomando como base el diseño de los modelos y no sólo su implementación, es posible automatizar la generación de esquemas de integración.
- Mantenimiento: Tener un diseño que sea entendido por una máquina brinda a los desarrolladores acceso directo a la especificación de los sistemas, haciendo del mantenimiento de las aplicaciones una tarea más simple.
- Pruebas y Simulación: Como los modelos desarrollados pueden usarse para generar código, pueden ser usados también para hacer validaciones con respecto a los requerimientos.

La iniciativa de MDE más popular es propuesta por el Object Management Group (OMG) y se conoce como MDA. MDA considera la generación de modelos específicos de la plataforma (PSM) a partir de modelos independientes de la plataforma (PIM) [23]. Esta característica aísla la lógica de la aplicación y las reglas de negocio, de la tecnología y las plataformas sobre las cuales se implementa el sistema. Como mecanismos básicos para apoyar esta estrategia el OMG propone el uso de modelos y transformaciones.

3.2.1. Modelos

Un modelo es una descripción de todo o parte de un sistema escrito en un lenguaje definido. Los modelos se pueden representar en 4 niveles de abstracción diferentes:

- Nivel M0 (instancia): son las instancias reales del sistema, es decir, los objetos de la aplicación.
- Nivel M1 (Modelo): representa el modelo de un sistema software, un modelo combina elementos y asociaciones. Los conceptos de M1 representan las instancias de M0.
- Nivel M2 (Metamodelo): es un modelo que describe modelos de M1. Un modelo es conforme a un metamodelo.
- Nivel M3 (Meta-metamodelo): los elementos de M3 representan las instancias de M2.

Entre los modelos y los metamodelos se establece la relación "conforme a", la cual indica que un modelo se encuentra especificado por el lenguaje definido en el metamodelo. Un ejemplo para ilustrar los niveles de abstracción es: UML es un metamodelo. Las modelos UML que se construyen para representar las aplicaciones son instancias del metamodelo UML. Si el modelo UML tiene la clase Canción, una instancia en M0 podría ser "Los caminos de la vida". UML se define usando MOF, MOF es un metametamodelo definido por el OMG.

3.2.2. Transformaciones

De forma general, en [23] se describe una transformación como el proceso de convertir un modelo en otro modelo del mismo sistema. Las transformaciones son construidas en lenguajes que cumplen la especificación QVT, ésta define los elementos que deberían estar presentes en un modelo de transformaciones.

Los enfoques de transformación que existen actualmente, pueden agruparse en dos grandes categorías [11]: "modelo a código" y "modelo a modelo". Dentro de la categoría de transformaciones modelo a modelo se encuentran los enfoques de manipulación directa (en los cuales los usuarios tienen que hacer prácticamente todo, y se les brinda poca o ninguna ayuda para la definición de las reglas así como el orden en que son aplicadas), los enfoques relacionales (en los que se utilizan relaciones matemáticas para establecer el tipo de relación entre los distintos elementos del modelo fuente y el modelo destino), los enfoques basados en transformaciones gráficas (que siendo los más poderosos también son los más complejos), los enfoques manejados por estructuras (en los que el usuario define las reglas de transformación y las herramientas se encargan de determinar el orden y estrategia de aplicación para estas), y los enfoques híbridos (en los que se usa una combinación de los enfoques anteriores) [11].

Los modelos origen y destino de una transformación son conformes a un metamodelo, la misma transformación es conforme a un metamodelo (ver figura 7).

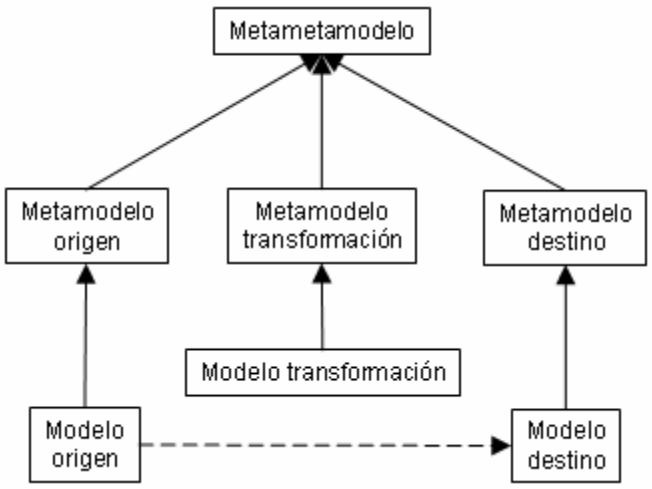

Figura 7. Modelo de transformación.

3.2.2.1. Modelo de transformación

Un modelo de transformación se conforma de un conjunto de transformaciones, las cuales especifican la forma en que los elementos del destino deben ser creados en términos del origen.

De acuerdo con [11] una transformación tiene dos partes: izquierda y derecha. La parte izquierda se encarga de acceder al modelo origen, mientras la parte derecha se encarga de hacer las operaciones necesarias sobre el modelo destino. Tanto la parte izquierda como la derecha pueden ser representadas mediante la utilización de las siguientes características:

Variables: Las variables contienen elementos del modelo origen o destino (o de elementos intermedios).
Lógica: La lógica expresa cálculos y restricciones sobre los elementos de los modelos. La lógica puede ser ejecutable o no ejecutable. Si es no-ejecutable se usa para especificar una relación entre los modelos. Por otro lado, cuando la lógica es ejecutable, puede ser de forma declarativa o imperativa.

En [20] se describe la diferencia entre los lenguajes declarativos y los imperativos. Un lenguaje es declarativo si sus reglas de transformación especifican las relaciones entre los elementos de los modelos origen y destino, sin involucrar un orden de ejecución. En contraste, cuando un lenguaje de transformación es imperativo, éste especifica una secuencia implícita de pasos para ser ejecutados

con el fin de producir un resultado. Puede existir una categoría intermedia, conocida como híbrida. En esta categoría los lenguajes tienen una mezcla de instrucciones declarativas e imperativas. Como se menciona en [20], una transformación escrita en este tipo de lenguajes puede tener un conjunto de reglas declarativas que describen las relaciones entre los modelos origen y destino. Adicionalmente cada regla, puede tener fragmentos de código imperativo, que realizan acciones adicionales para producir el resultado deseado.

Teniendo en cuenta que las transformaciones son una de las operaciones centrales de MDA se aplican al desarrollo de las mismas las buenas prácticas de la ingeniería de software tradicional.

En [21] se presentan varias técnicas de modularización para propiciar la reutilización de las transformaciones. La propuesta considera dos de ellas:

- Derivar las reglas de la correspondencia entre metamodelos, idealmente el mapeo entre metamodelo origen y metamodelo destino es de uno a uno.
- Aislar las partes de la transformación que no dependen de los metamodelos origen y metamodelos destino.

3.2.2.2. Patrones de transformación

El diseño de patrones en ingeniería de software debe capitalizarse como una buena práctica para incrementar la reutilización. MDA se beneficiaría al tener una colección de patrones de diseño para las transformaciones. En [6] se proponen dos patrones para diseñar transformaciones: parametrización y múltiples modelos origen.

Parametrización

Motivación. Algunos tipos de elementos del metamodelo destino no tienen correspondencia con tipos de elementos del metamodelo origen pero el diseñador desea colocarlos con cierto valor estáticos. Métodos que colocan los valores estáticos son codificados para ser llamados desde cualquier punto de la transformación.

Solución. Dentro de la solución proponen un metamodelo de parámetros, el cual se podría crear para cada transformación o definirse de forma genérica.

Consecuencias. Las transformaciones tienen que usar más de un modelo origen. El diseñador de la transformación debe especificar el modelo de parámetros.

Múltiples modelos origen

Motivación. Algunos lenguajes no soportan transformaciones que requieren más de un patrón origen.

Solución. Si el lenguaje de transformación no soporta varios patrones origen se podría usar precomputación: se combinan los elementos de los modelos origen involucrados usando una sentencia repetitiva y se coloca una expresión condicional que filtre los elementos de interés.

Consecuencias. El patrón origen se hace complejo pues se agregan expresiones imperativas.

3.2.3. Entrelazado

Además de las transformaciones, el entrelazado es otra operación importante en MDE. El objetivo del entretejido es especificar relaciones tipadas entre elementos de distintos modelos [13]. Las relaciones son capturadas en un modelo de entrelazado que es conforme a un metamodelo, el cual define los tipos de relaciones. El entrelazado se puede realizar a nivel de M1 o a nivel de M2, es decir, a nivel de modelos o de metamodelos. A diferencia de las transformaciones, las cuales son ejecutadas por un mecanismo de transformación, el entrelazado es hecho de forma manual por el usuario [14]. Cuando la operación de entrelazado es realizada se genera un tercer modelo, llamado modelo de entrelazado, el cual es conforme a un metamodelo de entrelazado.

Algunas aplicaciones del entretejido son:

Integración de modelos que necesitan interoperar. Por ejemplo pasar de un modelo de clases a un modelo relacional.

Diseño de las transformaciones entre metamodelos. Las correspondencias entre el metamodelo origen y el metamodelo destino se establecen de forma genérica con independencia del lenguaje de transformación.

Generación automática de transformaciones. A partir del diseño de las transformaciones se generan transformaciones en un lenguaje específico usando transformaciones de alto orden. Las transformaciones de alto orden (Higher Orden Transformation HOT) operan sobre transformaciones genéricas para producir transformaciones específicas [31].

Definir los tipos de relaciones en un metamodelo es la parte principal del entrelazado. A continuación se muestra un entrelazado entre un modelo de mundo y un modelo de arquitectura usando la relación “asociada a”, la cual debe

estar especificada en un metamodelo de entrelazado. En la figura 8 La relación "asociada a" representa que las clases Discotienda, Disco y Canción están asociadas a InterfazDiscotienda, PanelCrearDisco y PanelDatosCanciones respectivamente.

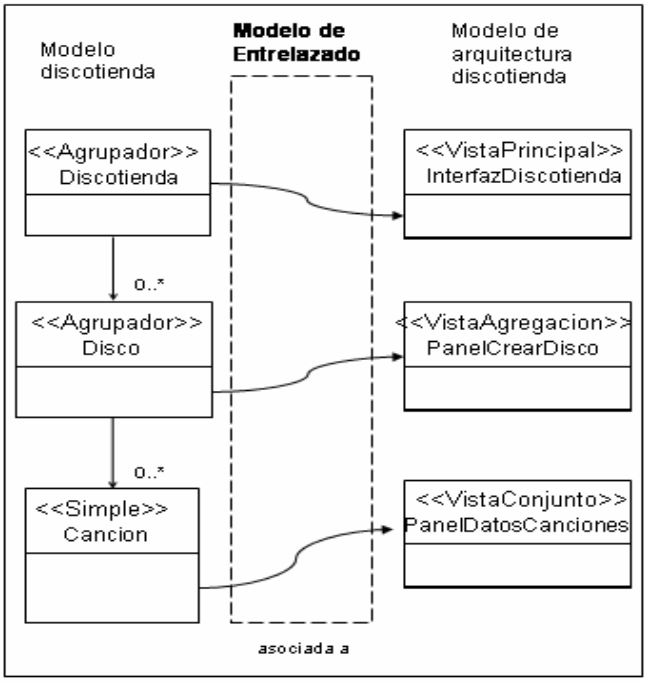

Figura 8. Modelo de entrelazado

3.3. MD-SPL

El objetivo de una MD-SPL es reducir el tiempo de desarrollo de los productos y mejorar la calidad y mantenibilidad de los mismos. MD-SPL es un programa de reutilización sistemática cuya tecnología está centrada en los modelos, los modelos son considerados artefactos reutilizables que sirven para representar formalmente los rasgos y la estructura de los sistemas [22].

La ingeniería del dominio y la ingeniería de la aplicación son procesos paralelos presentes en una MD-SPL. En la ingeniería del dominio se especifica la variación, es decir, se establecen los puntos de variación y los variantes. En la ingeniería de la aplicación la derivación del producto es guiada por la especificación de la variabilidad [22].

4. Solución propuesta

En la siguiente sección presentamos una estrategia para solucionar el problema de incluir las preferencias del usuario en los modelos destino al aplicar las transformaciones. Esta estrategia agrega al enfoque MD-SPL algunos elementos para expresar la variabilidad de los productos. Los elementos son modelos de rasgos, modelos de entrelazado y transformaciones. A continuación explicamos qué son y luego cómo se integran a los procesos del enfoque MD-SPL para generar modelos destino diferentes. La solución es ilustrada en la experimentación.

4.1. Modelo de rasgos

Los modelos de rasgos que se construyen representan las características comunes y variables de los modelos destino de una transformación. Los modelos de rasgos son conformes al metamodelo propuesto en [12] (ver Anexo A).

Si en el esquema de generación de un producto hay varias transformaciones, se construye un modelo de rasgos que caracterice los modelos destino de cada transformación. Los modelos de rasgos se pueden construir explorando el conjunto de modelos destino que se pueden generar, durante esta actividad se abstrae lo común y variable entre ellos.

4.2. Modelo de entrelazado entre el modelo origen y los rasgos del dominio destino

Para generar modelos destino ajustados a las preferencias del usuario, éste debe hacer una selección de aquellos rasgos que prefiere. La selección se hace asociando a los elementos del modelo origen de la transformación los rasgos. La selección es previa a la ejecución de la transformación, pues ésta se encarga de crear en el modelo destino elementos que satisfagan los rasgos seleccionados.
La operación que permite la selección es llamada entrelazado del modelo origen con rasgos. A través del entrelazado el usuario especifica en qué se transformarán los elementos del modelo origen, teniendo en cuenta las opciones ofrecidas por el modelo de rasgos. El entrelazado genera un modelo de entrelazado el cual contiene un conjunto de enlaces, cada enlace tiene dos partes una parte que hace referencia a un elemento del modelo origen y otra que hace referencia a un rasgo. El modelo de entrelazado es conforme a un metamodelo de entrelazado, una extensión del metamodelo propuesto en [13] (ver Anexo B). El usuario sólo debe hacer entrelazado entre los elementos del modelo origen y los rasgos opcionales y alternativos, el entrelazado con rasgos solitarios de tipo obligatorio no es necesario, pues se asume que dichos rasgos siempre deben ser satisfechos.

Cuando los rasgos son alternativos, el elemento del modelo origen sólo puede ser entrelazado con uno de ellos.

4.3. Transformaciones

Cuando la selección de rasgos ha sido realizada se ejecutan las transformaciones. La transformación tiene como entradas modelos de origen y modelos de entrelazado y como salida un modelo destino variable. Durante la ejecución de una transformación se aplican reglas de transformación base, de control y específicas, las cuales son construidas usando patrones como parametrización y precomputación propuestos en [6].

Para diseñar las reglas de transformación se relacionan los elementos de cada metamodelo origen y su respectivo modelo de rasgos destino. Así se identifican los elementos del metamodelo origen que se transforman siempre en los mismos elementos del metamodelo destino, para los cuales se crean reglas base, y los que pueden ser transformados de forma diferente (variable), para los que se crean reglas de control y reglas específicas.

Las reglas de transformación base generan lo común a los modelos destino. Las reglas de control y específicas adicionan lo variable a un modelo destino a partir de una selección de rasgos.

Una regla de transformación base, son implementadas de forma declarativa, tienen un patrón origen y un patrón destino, es decir, que cualquier elemento en el modelo origen, conforme al concepto definido en el patrón origen, será transformado automáticamente en un elemento, conforme al concepto definido en el patrón destino. La naturaleza de este tipo de reglas es aprovechada para generar lo común a los modelos destino.

Una regla de transformación específica, son implementadas de forma imperativa, es decir, son reglas que encapsulan una funcionalidad y que se pueden invocar varias veces. Este tipo de reglas tienen un nombre, puede recibir parámetros y puede contener un patrón destino y/o una secuencia de sentencias imperativas. Con las reglas específicas se logra reutilización y su funcionalidad está ubicada en una sola parte.

Una regla de transformación de control, tiene una parte declarativa y una parte imperativa. En nuestra propuesta las reglas de control son implementadas para manejar la variabilidad, ellas manejan las diferentes opciones que pueden seleccionarse durante el entrelazado. Una regla de control consulta el modelo de entrelazado en busca del rasgo asociado, en caso de encontrarlo, recupera el

elemento del modelo origen con el cual dicho rasgo está entrelazado. La regla de control crea elementos en el modelo destino y finalmente invoca reglas específicas para enriquecer los elementos creados de forma que se satisfaga el rasgo. Cuando una regla específica es invocada desde una regla de control, la regla específica configura los elementos del modelo destino a partir de los parámetros.

Habiendo descrito los elementos principales de esta propuesta, la integración de ellos a los procesos de una MD-SPL sería:

En el proceso de ingeniería del dominio se construyen los activos: un metamodelo de lógica de negocio, un metamodelo de arquitectura y un metamodelo de tecnología, reglas de transformación base, de control y específicas del modelo de lógica de negocio al modelo de arquitectura y de éste al modelo de tecnología, modelo de rasgos de la arquitectura y modelo de rasgos de la tecnología.

En el proceso de ingeniería de la aplicación el usuario debe proporcionar el modelo de lógica de negocio y entrelazar éste modelo con el modelo de rasgos de la arquitectura. Se aplica la transformación para generar el modelo de arquitectura. Generados los modelos de arquitectura el usuario debe entrelazar cada uno de estos con el modelo de rasgos de la tecnología. Se aplica la transformación para generar el modelo de tecnología y finalmente se aplica una transformación basada en plantillas para generar el código fuente (ver figura 9).

En los procesos descritos se distinguen dos roles: el desarrollador de los activos y el usuario que especifica sus preferencias para generar una aplicación determinada.

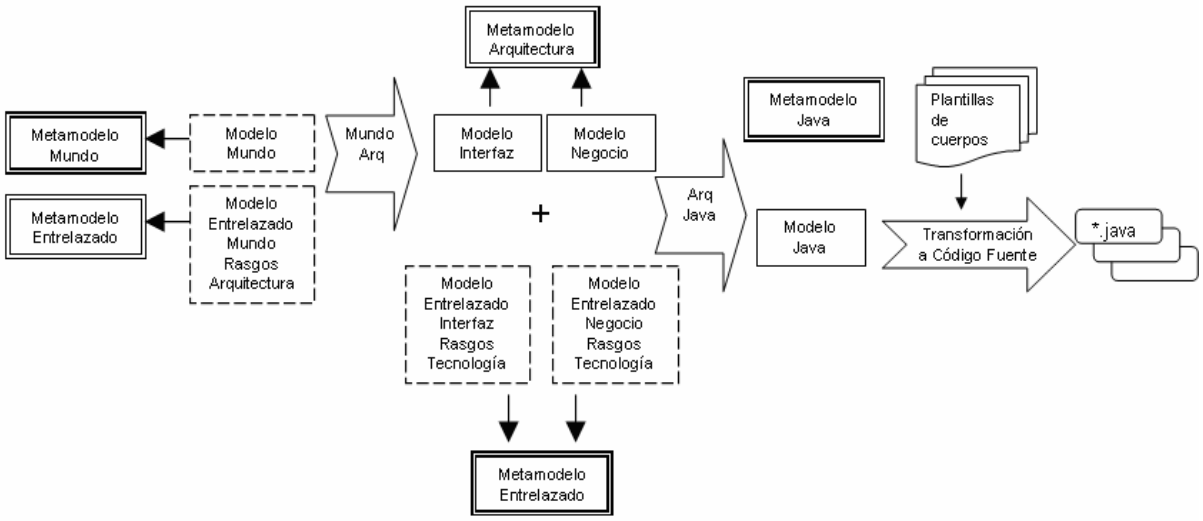

Figura 9. Proceso de creación de un producto con manejo de variabilidad.

5. Qualdev-MDSPL: Implementación de la solución

Qualdev-MDSPL es un plug-in para Eclipse que materializa la propuesta de solución. A través de este plug-in los usuarios pueden generar aplicación variables a partir de los activos de las líneas de producto basada en modelos. Qualdev-MDSPL se apoya en EMF y en los plug-ins AMW y ATL.

En las siguientes secciones se presenta la plataforma sobre la cual fue desarrollado Qualdev-MDSPL, se describen de forma general las funcionalidades que ofrece y se reseña su arquitectura.

5.1. Plataforma de desarrollo

Eclipse. Es una herramienta creada por IBM y OTI para el desarrollo integrado de aplicaciones [15]. Eclipse está basada en el esquema de plugins y construida sobre el modelo de componentes OSGI [24]. Un plug-in es un componente de software funcional que se puede conectar y desconectar de forma casi transparente.

EMF. Es un framework que facilita la construcción de modelos a través de una interfaz gráfica.

ATL. Es un lenguaje híbrido que permite implementar transformaciones declarativas, imperativas e híbridas, sin embargo el primer estilo es recomendado. Un programa de transformación ATL está compuesto por reglas que definen cómo los elementos del modelo origen son transformados en elementos del modelo destino.

Una regla declarativa especifica un patrón origen (source pattern) y un patrón destino (target pattern).

Una regla imperativa es básicamente un procedimiento, tiene un nombre, puede recibir parámetros y puede contener un patrón destino y/o una secuencia de sentencias imperativas (action block).

El patrón origen está compuesto por una colección de tipos de elementos de los metamodelos origen y una expresión booleana usada para filtrar los elementos.

El patrón destino está compuesto por una colección de tipos de elementos de los metamodelos destino, por cada tipo de elemento de esta colección hay un conjunto de bindings. Un binding especifica el valor de una propiedad del tipo de elemento destino usando una expresión.

Las reglas de transformación usan OCL para navegar por los modelos origen y destino. Durante la ejecución de una regla los modelos origen no pueden ser modificados y los modelos destino no pueden ser navegados.

AMW. Es un plug-in para Eclipse que permite hacer entrelazado entre modelos. La principal idea de la implementación es tener una interfaz de usuario simple que pueda adaptarse al metamodelo de entrelazado escogido por el usuario. La interfaz por defecto se ajusta al metamodelo de entrelazado base, tiene tres paneles: izquierdo, derecho y central. En el panel izquierdo se visualiza el modelo origen, en el panel derecho el modelo destino y en el panel central el modelo de entrelazado. El modelo de entrelazado se construye adicionando enlaces y arrastrando elementos de los modelos origen y destino que sean correspondientes.

5.2. Funcionalidades

5.2.1. Seleccionar una MD-SPL

Qualdev-MDSPL permite al usuario consultar y seleccionar una MD-SPL que se encuentre dentro de aquellas disponibles. Cuando una MD-SPL es escogida Qualdev-MDSPL carga el esquema de generación de la misma, el cual se define en un xml. El esquema de generación especifica las transformaciones secuenciales que deben ejecutarse, por cada transformación se define metamodelos origen, un metamodelo destino, y modelos de rasgos del modelo destino. Adicionalmente, Qualdev-MDSPL dispone todos los activos requeridos durante el proceso de generación de una aplicación: modelos, metamodelos y transformaciones.

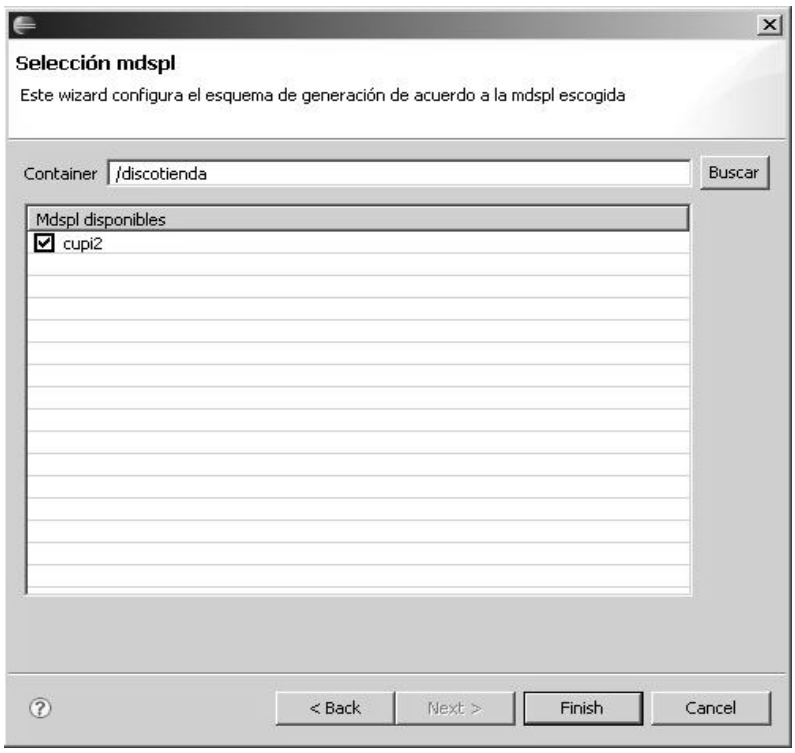

Figura 10. Seleccionando una MD-SPL en Qualdev-MDSPL.

5.2.2. Crear modelos origen

Qualdev-MDSPL permite al usuario crear modelos origen que sean entradas al proceso de transformación. Para lograrlo utiliza EMF.

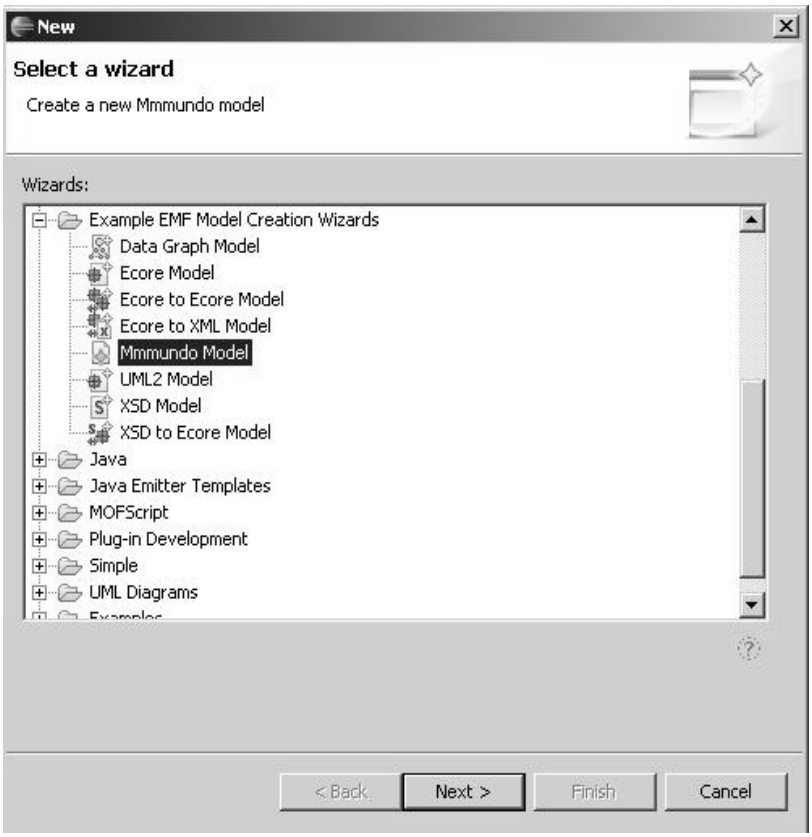

Figura 11. Creando un modelo origen

5.2.3. Especificar preferencias sobre un modelo destino

Qualdev-MDSPL permite que el usuario especifique sus preferencias sobre un modelo origen antes de aplicar una transformación. Esto es posible a través de la interfaz gráfica de usuario del plug-in AMW, en donde se entrelazan los elementos del modelo origen con los rasgos del modelo destino. El modelo origen es creado por el usuario o puede resultar de una transformación anterior. El modelo de rasgos con el cual debe entrelazarse se define en el esquema de generación. La información de entrelazado se persiste en un modelo de entrelazado.

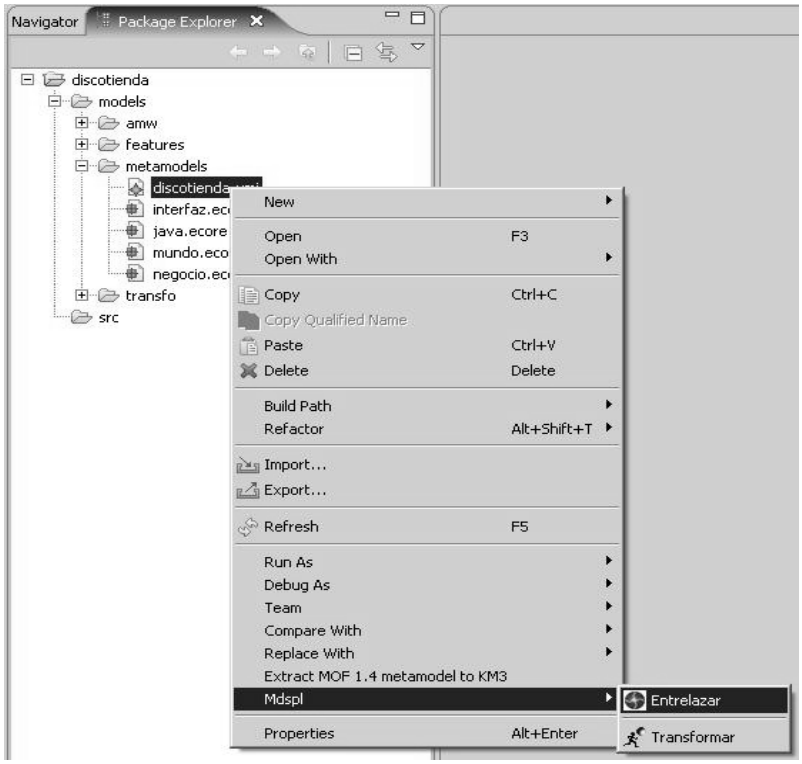

Figura 12. Escogiendo el modelo origen a entrelazar con los rasgos del modelo destino

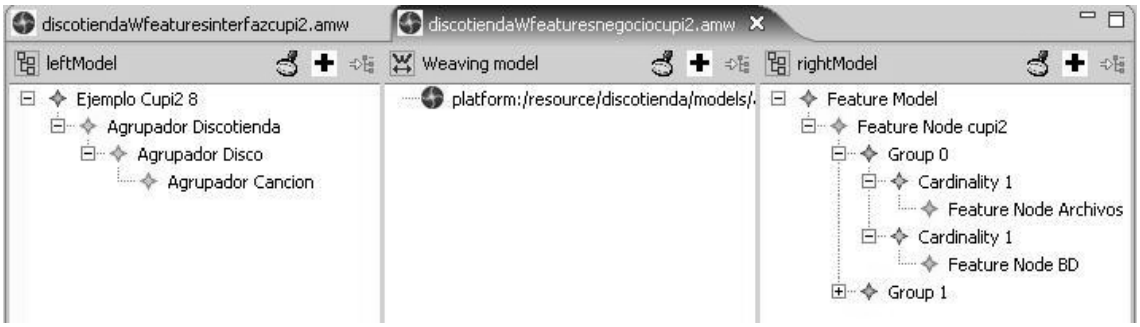

Figura 13. Haciendo entrelazado entre un modelo origen y un modelo de rasgos del destino

5.2.4. Generar un modelo destino

Qualdev-MDSPL permite que el usuario genere un modelo destino a partir de modelos origen y modelos de entrelazado entre modelo origen y rasgos del modelo destino. Internamente Qualdev-MDSPL invoca el plug-in ATL para que ejecute la transformación adecuada dado el modelo origen. La transformación tiene como entradas un metamodelo origen, un modelo origen, un metamodelo de entrelazado, un modelo de entrelazado y como salidas un modelo destino, que es conforme a un metamodelo destino. Esta información se encuentra definida en el esquema de generación.

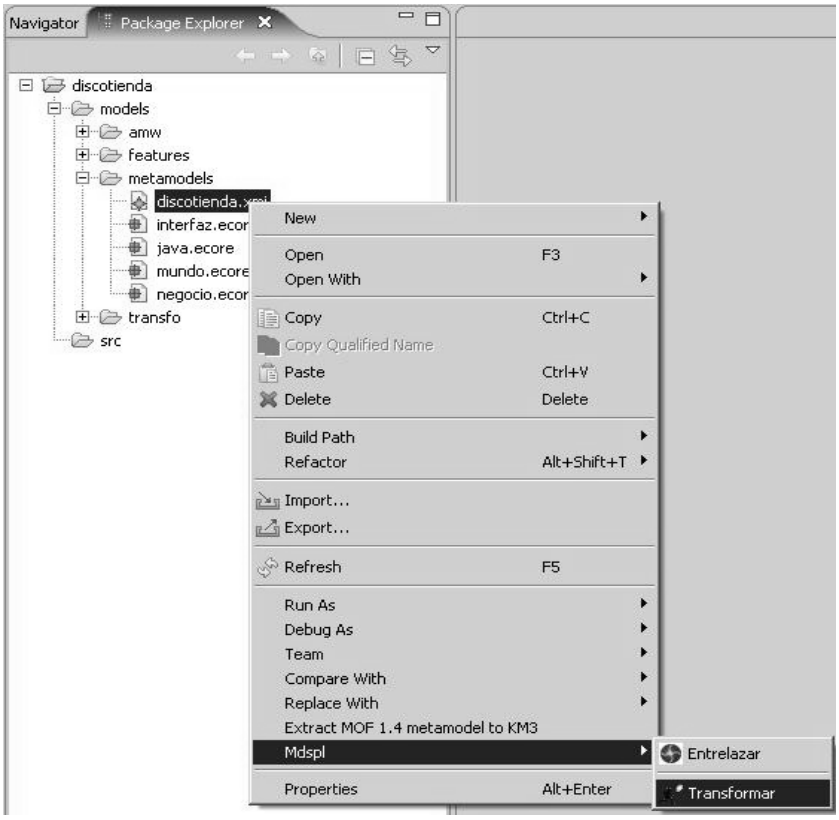

Figura 14. Escogiendo el modelo origen de la transformación

Para llegar al último modelo destino, operaciones de entrelazado y transformación deben ejecutarse de acuerdo a lo definido en el esquema. Cuando se tiene el modelo destino el usuario debe invocar la generación del código a través de plantillas.

5.3. Arquitectura

Qualdev-MDSPL implementa todas las funcionalidades a través de dos capas independientes pero complementarias: capa de negocio y capa de interfaz gráfica (ver figura 15).

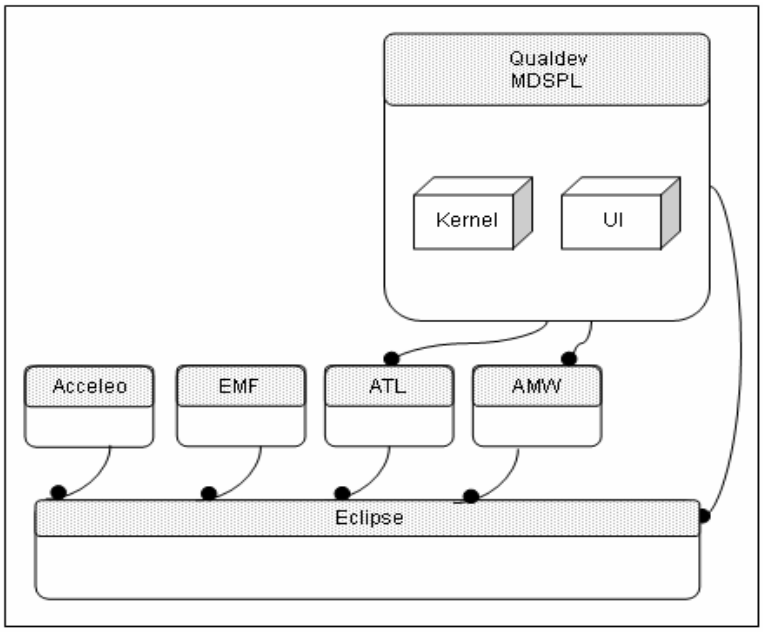

Figura 15. Arquitectura del plug-in Qualdev-MDSPL

5.3.1. Capa de negocio

Está conformada por cuatro componentes:

MdsplResouces. Provee información sobre las MD-SPL disponibles.
MdsplParser. Procesa el xml de configuración de la MD-SPL escogida y determina los activos que deben utilizarse en las operaciones de entrelazado y transformación.
MdsplManager. Controla el esquema de generación de la MD-SPL escogida. Invoca AMW y ATL con los modelos y metamodelos definidos en el esquema.
Util. Implementa funcionalidad común a los demás componentes.

5.3.2. Capa de interfaz gráfica

Comprende todas las clases necesarias para visualizar y manejar el plug-in desde Eclipse.

6. Experimentación

La experimentación de la solución propuesta fue hecha en un subconjunto de las aplicaciones Cupi2. La experimentación retoma los ejemplos Exposición de vehículos (nivel 7) y Discotienda (nivel 8).

6.1. Objetivos

- Evaluar la correctitud de la solución propuesta para manejar variabilidad en MD-SPL.

6.2. Ejecución de la experimentación

6.2.1. Ingeniería del dominio

Para realizar la experimentación se partió de los metamodelos desarrollados en [25]: metamodelo de mundo, arquitectura (negocio e interfaz) y java. Se desarrollaron en forma conjunta con Carlos Parra modelos de rasgos, modelos de entrelazado, transformaciones base, de control y específicas [25].

Se construyeron modelos de rasgos por cada uno de los modelos destino. Un modelo de rasgos de negocio, uno de interfaz y otro de java. Los modelos contienen las características comunes y variables identificadas en [25]. Recordar que las características comunes y variables manejadas son un subconjunto de las características del dominio Cupi2. Las características manejadas pertenecen a las aplicaciones de los niveles 3 al 9. En la figura 16 se aprecian los modelos en un diagrama jerárquico, una descripción detallada de ellos se encuentra en el Anexo C.

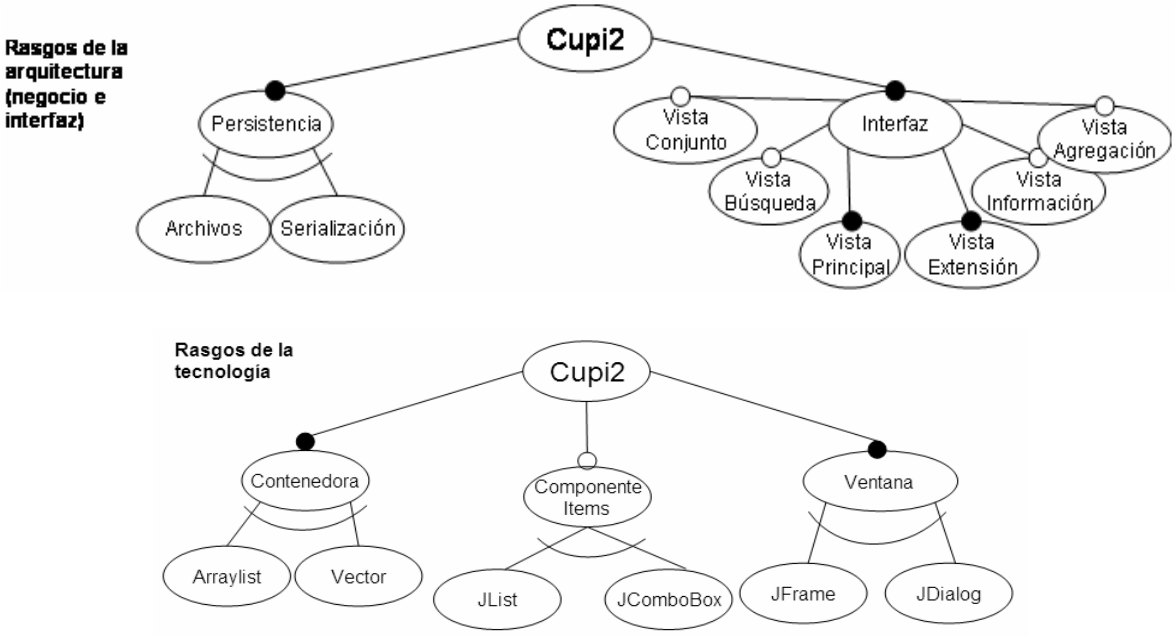

Figura 16. Modelo de rasgos de la arquitectura y de la tecnología en Cupi2.

El código de las transformaciones desarrolladas se puede consultar en los anexos de [25]. Cada uno de los activos mencionados fue utilizado por Qualdev-MDSPL durante la generación de los ejemplos.

6.2.2. Ingeniería de la aplicación

Para desarrollar los ejemplos discotienda y exposición automóviles se desarrollaron modelos de origen, modelos de entrelazado y se ejecutaron transformaciones.

6.2.2.1. Modelos origen

Utilizando los wizards de creación de modelos instalados en Qualdev-MDSPL se crean dos modelos conformes al metamodelo de mundo, uno corresponde al ejemplo discotienda y otro al ejemplo exposición de vehículos.

6.2.2.2. Modelos de entrelazado

Qualdev-MDSPL invoca la interfaz AMW para realizar las operaciones de entrelazado sobre cada modelo de mundo. En MD-SPL Cupi2 el primer entrelazado se efectúa entre el modelo de mundo y los rasgos de arquitectura. Para efectos de la ilustración el modelo de entrelazado se muestra gráficamente,

en lugar de mostrarlo en el formato xmi que genera AMW. Se muestran los modelos de entrelazado del ejemplo Discotienda y luego los del ejemplo Exposición Vehículos.

Entrelazados Discotienda

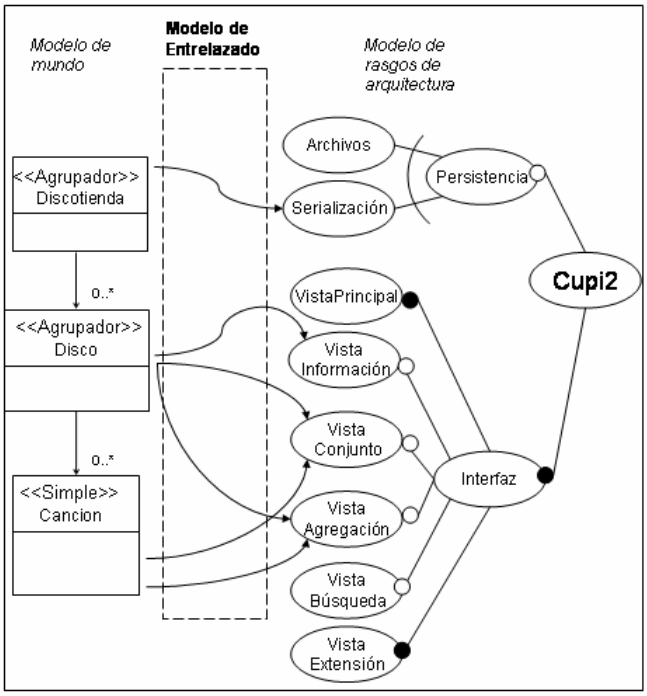

Figura 17. Entrelazado entre el modelo de mundo de discotienda y los rasgos de arquitectura.

La figura 17 muestra enlaces que tienen como origen elementos del modelo de mundo de discotienda y como destino rasgos de la arquitectura.
El enlace entre el elemento Discotienda y el rasgo Serialización indica que para Discotienda y todos los elementos contenidos (Disco y Canción) se generarán clases que implementan la interfaz Serializable. Adicionalmente se implementa un método que guarda en un archivo binario los datos de discotienda, discos y canciones. Los enlaces entre Disco y VistaInformación, VistaConjunto y VistaAgregación y los enlaces entre Canción y VistaConjunto, VistaAgregación indican que se generarán clases que implementan JPanel y que contienen los componentes gráficos propios de cada vista.

Entrelazados Exposición Vehículos

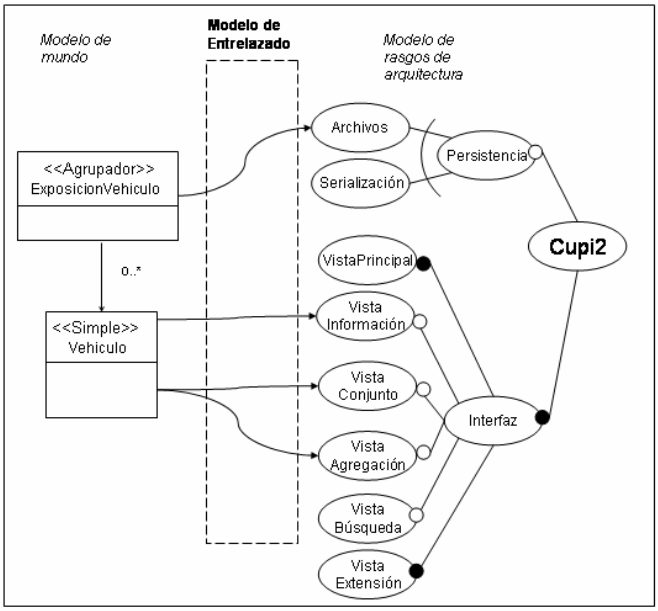

Figura 18. Entrelazado entre el modelo de mundo exposición vehículos y los rasgos de arquitectura.

La figura 18 muestra enlaces que tienen como origen elementos del modelo de mundo de Exposición Vehículos y como destino rasgos de la arquitectura.
Para satisfacer el enlace entre el elemento ExposicionVehiculo y el rasgo Archivos se implementarán métodos para escribir/leer en/de un archivo los datos de exposición vehículos y vehículos. La implementación es hecha con las clase PrintWriter y BufferedReader. Note aquí una variación con el ejemplo discotienda, donde la persistencia se hace a través de serialización. Los enlaces entre Canción y VistaInformación, VistaConjunto y VistaAgregación tienen el mismo efecto descrito en el Modelo de entrelazado entre el modelo de mundo y los rasgos de arquitectura del ejemplo discotienda. La diferencia está en los elementos del modelo de mundo y las vistas entrelazadas.

6.2.2.3. Transformación

Cuando se finaliza el entrelazado se ejecuta la transformación mundo – arquitectura, para tal efecto Qualdev-MDSPL invoca ATL. A continuación se presentan algunas reglas de transformación usando ATL. No se presenta una explicación detallada de la sintaxis ATL, sino que se ha adaptado para propósitos de la ilustración. Cabe recordar que las reglas de transformación que generan lo común a los modelos destino son base y las reglas que manejan la variabilidad especificada en los modelos de entrelazado son de control y específicas.

Regla base

```
1. rule vistaPrincipal {
2.         from
3.                 a : Agrupador  (a.esPrincipal = true)
4.         to
5.                 v : VistaPrincipal()
6.  }
```

Figura 19. Regla de transformación base

La figura 19 muestra una regla de transformación base. En la regla vistaPrincipal el patrón origen se define en las líneas (2-3) y el patrón destino en las líneas (4-6). La regla vistaPrincipal consulta el modelo de mundo en busca de los elementos conformes al concepto Agrupador (línea 3) que tengan el atributo esPrincipal en verdadero, de encontrarse alguno, se crea una Vista Principal (línea 5). La vista principal es común a todos los ejemplos de la línea por lo tanto siempre se utiliza esta regla de transformación. Si se tiene como modelo de mundo la tienda de discos, se crea una Vista principal a partir del elemento Discotienda. Si se tiene como modelo de mundo la Exposición de vehículos, se crea una Vista principal asociada a ExposiciónVehículo.

Regla de control

```
17. rule vistaAgrupador{
18.  from
19.     e : Enlace(e.origen = Agrupador)
20. do{
21.   if(e.rasgo = VistaConjunto){
22.              addVista(#lista);
23.   }
24.   if(e.rasgo = VistaInformacion){
25.              addVista(#etiqueta);
26.   }
27.  }
28. }
```

Figura 20. Regla de control

En la figura 20, se muestra una regla de control, en las líneas 18-19 se define la parte declarativa y en las líneas 20-28 la parte imperativa. La regla vistaAgrupador consulta el modelo de entrelazado, si en el modelo origen existe un elemento Agrupador, se manejan dos situaciones: si el rasgo con el cual esta entrelazado el elemento es Vista Conjunto se creará una vista con los elementos propios de una vista conjunto (líneas 21-23), si el rasgo con el cual esta entrelazado el elemento es Vista Información se creará una vista con los elementos propios de una vista conjunto (líneas 24-26). Al revisar el modelo de entrelazado de discotienda, se encuentra un enlace Disco - VistaConjunto y un enlace Disco – Vista Información, lo cual hace que se creen dos vistas, una conjunto y otra información asociadas a Disco. La creación de las vistas se realiza invocando a la regla específica addVista (línea 22 y 26).

Regla específica

```
7.  rule addVista (t: Tipo){
8.    to
9.      c : Visualizacion(
10.                tipo <- t
11.     )
12.     v : Vista(
13.       atributo.add(c)
14.     )
15. }
```

Figura 21. Regla de transformación específica.

La figura 21 muestra la regla imperativa addVista, en la línea 7 se aprecia el nombre de la regla y sus parámetros, en las líneas 8-15 se aprecia el patrón destino. La regla addVista crea un componente de tipo Visualizacion (línea 9), le asigna el tipo que recibe por parámetro (línea 10), luego crea una Vista (línea 12-14) y le agrega a ésta el componente de visualización previamente creado. La regla addVista puede invocarse para que cree un componente en una vista particular (etiquetas, cuadros de texto, listas, entre otros.), las invocaciones pueden hacerse de acuerdo a la necesidad. La regla de control presentada anteriormente invoca addVista, para una VistaConjunto agrega una lista y para una VistaInformación agrega etiquetas.

Al ejecutar la transformación se obtienen los modelos de arquitectura, entre estos modelos y los rasgos de tecnología se realizan nuevamente operaciones de entrelazado (ver Anexo D). Cuando se finaliza la operación se ejecuta una transformación de arquitectura a tecnología la cual usa patrones similares a los aplicados en la transformación mundo-arquitectura. Cuando se ha generado el modelo de java se aplican las plantillas las cuales generan el código fuente. Las interfazs gráficas de usuario de las aplicaciones generadas se aprecian en la figuras 22 y 23.

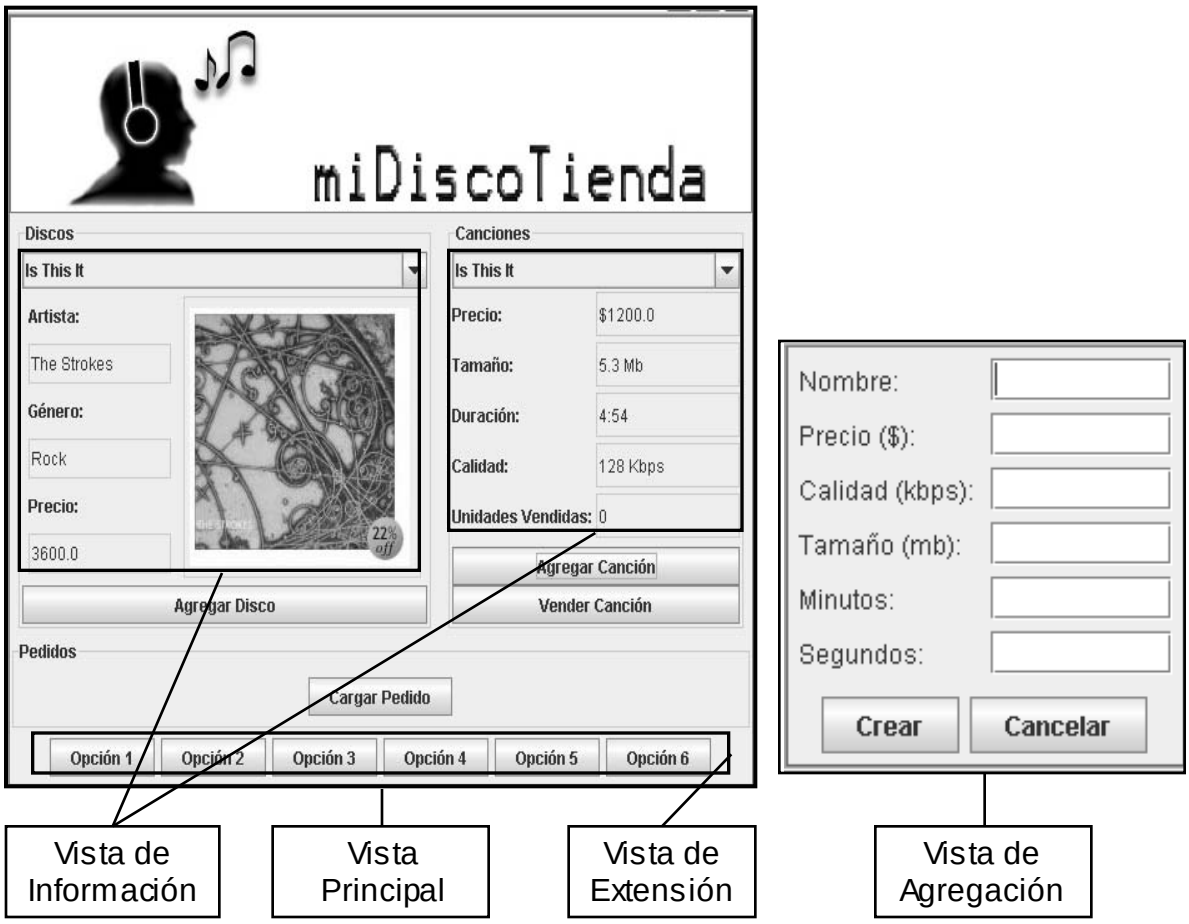

Figura 22. Interfaz gráfica de las aplicación generada para el ejemplo discotienda.

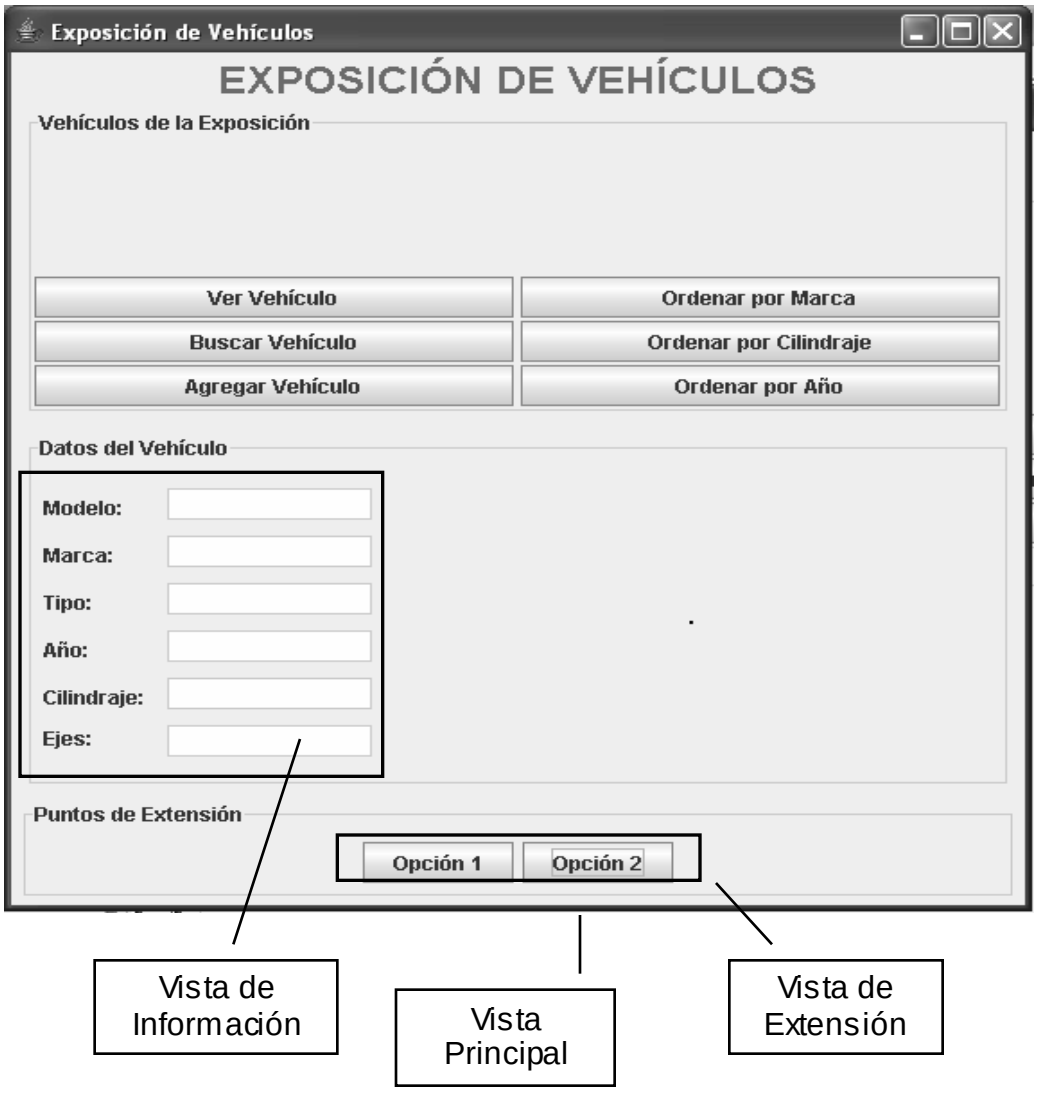

Figura 23. Interfaz gráfica de las aplicación generada para el ejemplo exposición vehículos.

6.3. Resultados obtenidos

Se puede afirmar que la solución propuesta permite manejar la variabilidad en una línea de productos basada en modelos.

En la experimentación se indican preferencias diferentes sobre dos modelos origen conformes al mismo metamodelo (discotienda y exposición automóviles), luego al aplicar incrementalmente las transformaciones se generan modelos destino diferentes, sin necesidad de modificar las transformaciones pues ellas manejan las variantes posibles. El resultado del proceso de generación son dos ejemplos Cupi2 con los componentes de mundo e interfaz diferentes.

La generación con Qualdev-MDSPL requiere el conocimiento del esquema de generación de la MD-SPL escogida, el usuario debe seguir este esquema de forma estricta para que no se generen errores. Adicionalmente el usuario debe conocer ciertas suposiciones y restricciones que se establecen sobre el entrelazado, para que no se haga entrelazado redundante o se generen modelos con elementos inesperados (ver anexo F).

El entrelazado es manual, por lo tanto los modelos de rasgos que se entrelazan deben ser minimales de forma que dicha operación no se convierta en algo tedioso.

7. Aportes de la investigación

De acuerdo con los resultados obtenidos en la experimentación y con los elementos estructurales de la solución, los principales aportes de la investigación realizada son:

7.1. Identificación de formas de variabilidad en MD-SPL

Se identificaron dos formas de generar modelos destino variables en líneas de producto basadas en modelos. Por un lado se encuentra la variabilidad que se puede expresar al construir el modelo origen a partir del cual se construye el modelo destino y por otro se encuentra la variabilidad que puede aplicar la transformación al modelo destino. La variabilidad a nivel de los modelos es manejada naturalmente, la variabilidad a nivel de las transformaciones requiere un tratamiento especial.

7.2. Propuesta para manejar variabilidad en una línea de productos basada en modelos

Se presentó una propuesta que permite manejar variabilidad en líneas de producto basadas en modelos. Los modelos de rasgos permiten expresar las características de una aplicación en un dominio destino específico (arquitectura, tecnología). Los metamodelos permiten crear un amplio conjunto de modelos variables en un dominio particular ampliando el alcance de la SPL. Las reglas de transformación base permiten crear aplicaciones con las características comunes de la línea. Las reglas de control y específicas permiten crear aplicaciones con las características variables de la línea. Finalmente, las operaciones de entrelazado entre modelos origen de una transformación y modelos de rasgos del dominio destino de la transformación permiten hacer configuración de las aplicaciones en los diferentes dominios, la configuración es materializada luego durante la ejecución automática de reglas que generan modelos. A partir de modelos variables se obtienen aplicaciones variables.

7.3. Implementación Qualdev-MDSPL

Se construyó un plug-in para Eclipse llamado Qualdev-MDSPL, el cual facilita la generación de aplicaciones miembros de una MD-SPL particular. El plug-in se apoya en los plug-ins EMF, AMW y ATL para crear los modelos, realizar entrelazado y ejecutar transformaciones.

7.4. Experimentación de la solución propuesta en el proyecto Cupi2

La solución propuesta se incorporó al trabajo desarrollado en [25]. Se definieron e implementaron modelos de rasgos, modelos de entrelazado y transformaciones para generar algunas características de los ejemplos Cupi2 en los niveles del 3 al 9.

8. Trabajos futuros

8.1. Restricciones sobre el entrelazado

Las restricciones sobre el entrelazado se expresaron en lenguaje natural. Es necesario un mecanismo que permita expresar y validar las restricciones de forma que el entrelazado sea más eficiente.

8.2. Semi-automatización del entrelazado

Las operaciones de entrelazado que se realizan antes de ejecutar una transformación entre modelos son hechas de forma manual por el usuario. Para hacer del entrelazado una operación más amigable se podrían utilizar algoritmos o heurísticas que lo semi-automaticen.

8.3. Aplicación de la propuesta en otros dominios

La solución propuesta fue aplicada en el dominio del proyecto Cupi2. En particular se desarrollaron activos para generar ejemplos con algunas de las características requeridas por los niveles del 3 al 9, sin embargo se podrían desarrollar nuevos activos para generar ejercicios o para cubrir más niveles de los ejemplos. La propuesta también podría ser aplicada en dominios diferentes a Cupi2.

8.4. Variabilidad a nivel de los requerimientos funcionales

La propuesta presentada maneja variabilidad relacionada con la estructura de los modelos, sin embargo los modelos pueden tener funcionalidades asociadas, manejar la variabilidad en dichas funcionalidades se dimensiona como un trabajo futuro.

8.5. Trazabilidad en líneas de producto basadas en modelos

La trazabilidad es tema importante dentro del proceso de desarrollo y por tanto es concerniente a las líneas de producto basadas en modelos. La trazabilidad se refiere a mantener enlaces entre los artefactos de un proceso de desarrollo, de forma que actividades como el análisis de impacto de cambios o mantenimiento se faciliten. En una MD-SPL sería deseable controlar la propagación de los cambios de un modelo sobre los demás modelos.

9. Conclusiones

Las líneas de producto basadas en modelos pretenden la generación de aplicaciones con características comunes y diferentes a partir de activos como metamodelos, modelos y transformaciones. La generación de productos diferentes requiere el manejo de la variabilidad. Este trabajo de investigación presentó una propuesta para manejar variabilidad en MD-SPL. La separación de dominios en metamodelos permite ampliar el alcance de la SPL administrando la variabilidad a nivel de conceptos de cada dominio de forma independiente. Los modelos de rasgos además de abstraer la variabilidad de las aplicaciones en los diferentes dominios, son empleados durante la generación de un producto para que el usuario a través de entrelazado indique sus preferencias sobre los modelos origen. Las preferencias se materializan luego en los modelos destino a partir de transformaciones base, de control y específicas. A partir de modelos destino variables se obtienen productos diferentes.

Dentro del trabajo de investigación se desarrolló un plug-in de Eclipse que facilita la generación de aplicaciones personalizadas usando modelos. El plug-in fue utilizado en la experimentación de la solución propuesta, la cual se llevó a cabo en el dominio de Cupi2.

Con el enfoque de líneas de producto basadas en modelos se obtienen varios beneficios. Por un lado con los modelos se amplia el alcance de una línea de producto y se logra la separación de preocupaciones y por otro con el enfoque de línea se mejora la productividad en el proceso de desarrollo pues los activos se reutilizan en la generación de productos diferentes.

La propuesta presentada sigue el principio MDE “todo es un modelo” ya que utiliza para solventar el problema de administrar la variabilidad modelos de rasgos, modelos de entrelazado y modelos de transformación. Adicionalmente introducir modelos de rasgos en la propuesta provee los beneficios de usar un estándar de facto. Llegar a una estrategia para administrar variabilidad comprometió muchos esfuerzos y en el camino se desecharon varias propuestas.

Los resultados obtenidos como parte de la implementación de la propuesta revelan un amplio número de puntos para trabajar: construir reglas de transformación más modulares aplicando patrones de diseño, mejorar el proceso de selección de rasgos manejando las restricciones entre los modelos de rasgos, expresar requerimientos funcionales a nivel de modelos e incorporarlos al esquema de generación, controlar la propagación de los cambios de un modelo sobre los demás modelos.

10. Bibliografía

[1] Alves, V., Dantas, A., Borba, P., AOP-Driven Variability in Product Lines of Pervasive Computing Applications en *Generative Programming and Component Engineering Conference*, Erfurt - Alemania, 2003.

[2] Arboleda, H., Casallas, R. QualDev-M: Línea de Producto Orientada por Modelos, *Universidad de Los Andes,* Bogotá- Colombia*,* 2006.

[3] Batory, D., Neal J., Scaling Step-Wise Refinement en *International Conference Software Engineering*, Portland - USA, 2003.

[4] Batory, D., Lopez-Herrejon R., Martin J., Generating Product-Lines of Product Families en *Automated Software Engineering Conference*, Texas – USA, 2002.

[5] Batory, D., Feature-Oriented Programming and the AHEAD Tool Suite en *International Conference on Software Engineering*, Texas - USA, 2004.

[6] Bézivin, J., Jouault, F., Towards model transformation design patterns en *Proceedings of the First European Workshop on Model Transformation*, Rennes - Francia, 2005.

[7] Cupi2. En línea: http://cupi2.uniandes.edu.co, Última visita Diciembre 2006

[8] Czarnecki, K, Antkiewicz, M. Mapping Features to Models: A template approach based on superimposed variants en *Generative Programming and Component Engineering Conference*, Tallinn - Estonia, 2005.

[9] Czarnecki, K., Helsen, S., Eisenecker, U., Formalizing Cardinality-based Feature Models and their Specialization en *Software Process Improvement and Practice Journal,* 2005

[10] Czarnecki, K., Helsen, S., Eisenecker, U., Staged Configuration Using Feature Models en *Software Product Line Conference*, Boston - USA, 2004.

[11] Czarnecki K, Helsen S. Classification of Model Transformation Approches en *Object-Oriented Programming, Systems, Languages and Applications Conference,* California - USA, 2003.

[12] Czarnecki, K., Model driven architecture. *University of Waterloo*. West Waterloo – Canada, 2004.

[13] Didonet Del Fabro, M, Jouault, F, van den Berg, K. Model Transformation and Weaving in the AMMA Platform en *Generative and Transformational Techniques in Software Engineering*, Braga - Portugal, 2005.

[14] Didonet Del Fabro, M, Bézivin, J. AMW: A Generic Model Weaver en *Premières Journées sur l'Ingénierie Dirigée par les Modèles*, Paris - Francia, 2005.

[15] Eclipse. En línea http://www.eclipse.org. Ultima visita Enero 2007´.

[16] Gomaa, H. Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures. Ed. AddisonWesley, 2004.

[17] Kang, K, Lee, J. Feature-Oriented Product Line Engineering. *IEEE Software*. New York – USA, 2002.

[18] Kang K, et al. Feature-Oriented Domain Analysis (FODA): Feasibility Study. *ACM Computing* Surveys. New York - USA, 1990.

[19] Kang K., Kim S., FORM: A feature-oriented reuse method with domain-specific reference architectures. *Ann. Softw. Eng.* New York – USA, 1998.

[20] Kurtev I. Adaptability of Model Transformations. *University of Twente*, Nederlands – Alemania, 2005.

[21] Kurtev, I., Jouault, F., Evaluation of Rule-based Modularization in Model Transformation Languages illustrated with ATL en *Symposium on Applied Computing*, Dijon - Francia, 2006.

[22] MBSE. Model based software engineering. En línea: http://www.sei.cmu.edu/mbse/is.html. Ultima visita Enero 2007

[23] OMG. MDA Guide Versión. 2003.

[24] OSGI Technology. En línea http://www.osgi.org/osgi_technology/. Ultima visita Enero 2007.

[25] Parra, C. Cupi2: una línea de productos basada en modelos. *Universidad de los Andes*, Bogotá – Colombia, 2007.

[26] Pérez, Danilo. Herramientas de apoyo a la enseñanza de programación bajo una perspectiva de líneas de producción de software. *Universidad de los Andes,* Bogotá – Colombia, 2006.

[27] Sametinger, J. Software engineering with reusable components. Springer, New York - USA, 1997.

[28] SEI. Software Product Lines. En Línea: http://www.sei.cmu.edu/productlines/, Ultima visita Diciembre 2006.

[29] Schmidt, D., Model-Driven Engineering. *Vanderbilt University.* Tenessee – Canada, 2006.

[30] Villalobos, J., Casallas, R., Marcos K. El Reto de Diseñar un Primer Curso de Programación de Computadores en *XIII Congreso Iberoamericano de Educación Superior en Computación,* Cali - Colombia, 2005.

[31] Varró, D., Pataricza, A., Generic and Meta-Transformations for Model Transformation Engineering en *Proceedings UML 2004: 7th International Conference on the Unified Modeling Language*, Lisboa - Portugal, 2004.

11. Anexo A. Metamodelo de rasgos

Los modelos de rasgos son conformes al metamodelo propuesto en [12]. En la figura 24 se aprecia dicho metamodelo.

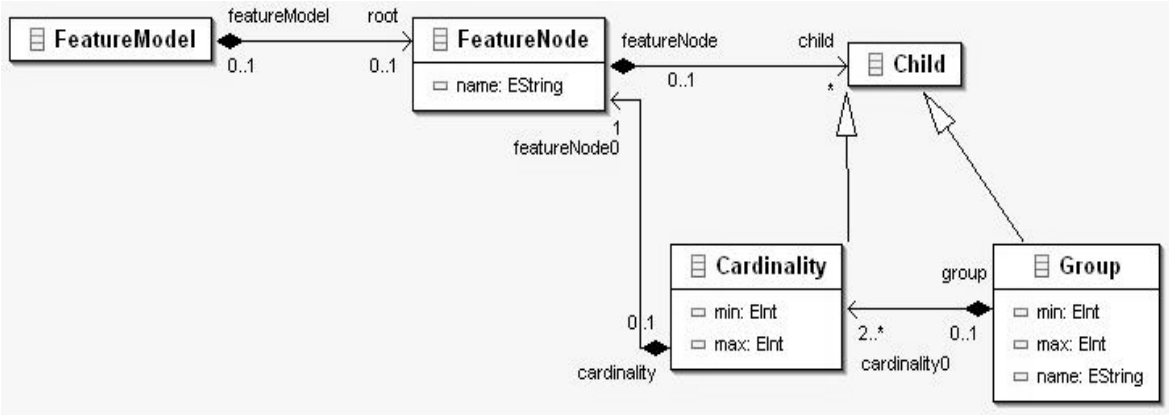

Figura 24. Metamodelo de rasgos

Una descripción de los elementos del metamodelo es:

FeatureModel. Metaclase de entrada al metamodelo de rasgos. Está compuesto por un rasgo llamado raíz.

FeatureNode. Representa un rasgo el cual se identifica con un nombre. De un rasgo puede derivarse otros rasgos llamados hijos.

Child. Representa los rasgos hijos que pueden derivarse de un rasgo. Un rasgo hijo puede ser de dos tipos Group ó Cardinality.

Cardinality. Representa un rasgo solitario. Si los atributos min - max tienen los valores 1-1 se asume que el rasgo es obligatorio. Si los atributos min - max tienen los valores 0-1 se asume que el rasgo es opcional. Cardinality tiene asociado un FeatureNode que permite representar la naturaleza jerárquica de los rasgos.

Group. Agrupa dos o más rasgos de tipo Cardinality. Si los valores min – max tienen los valores 1-1 se asume que los rasgos agrupados son alternativos entre sí. Group tiene un nombre que lo identifica.

12. Anexo B. Metamodelo de entrelazado

El metamodelo de entrelazado sugerido en la solución propuesta, es una extensión del metamodelo propuesto en [14]. El metamodelo de entrelazado se aprecia en la figura 25.

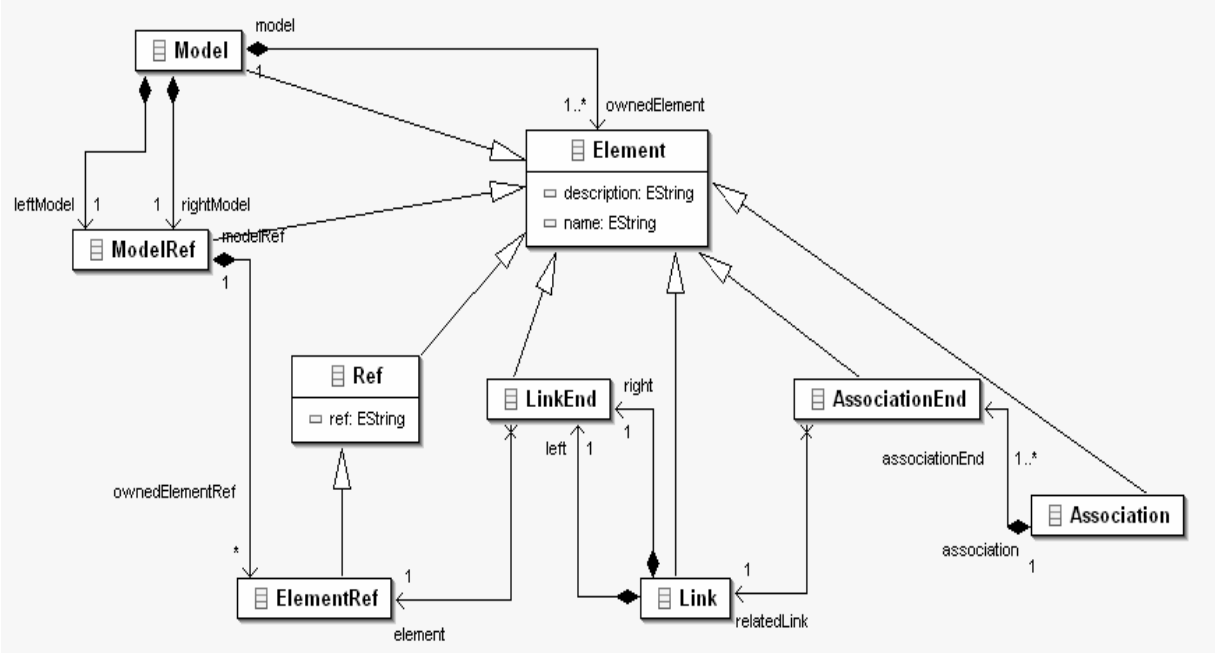

Figura 25. Metamodelo de entrelazado

Los elementos del metamodelo son:

Element. Es el elemento base de todos los elementos del metamodelo. Los demás elementos extienden de él. Tiene dos atributos name y description.

Model. Elemento raíz del metamodelo de entrelazado. Está compuesto de Links y de referencias a los modelos entrelazados, una referencia para el modelo de la izquierda y otra referencia para el modelo de la derecha.

Link. Representa un enlace entre elementos de los modelos entrelazados. La referencia left y right permite entrelazar sólo un elemento del modelo de la izquierda y del modelo de la derecha respectivamente.

LinkEnd. Indica la extremidad de un link, referencia los elementos del modelo entrelazado a través de ElementRef.

Ref. Clase abstracta que representa las referencias.

ElementRef. Todos los elementos referenciados de un modelo entrelazado. El atributo ref contiene el identificador de los elementos entrelazados.

ModelRef. Referencia un modelo que está siendo entrelazado. Está compuesto de referencias a los elementos de dicho modelo.

Association. Usado para crear relaciones entre enlaces.

AssociationEnd. Similar a LinkEnd, especifica las extremidades de una Association.

13. Anexo C. Modelos de rasgos

Los modelos de rasgos fueron representados siguiendo la especificación FODA, es decir, una descripción textual y un diagrama jerarquizado.

La figura 26 muestra los modelos de rasgos empleados en la experimentación. Seguidamente se encuentra la descripción textual.

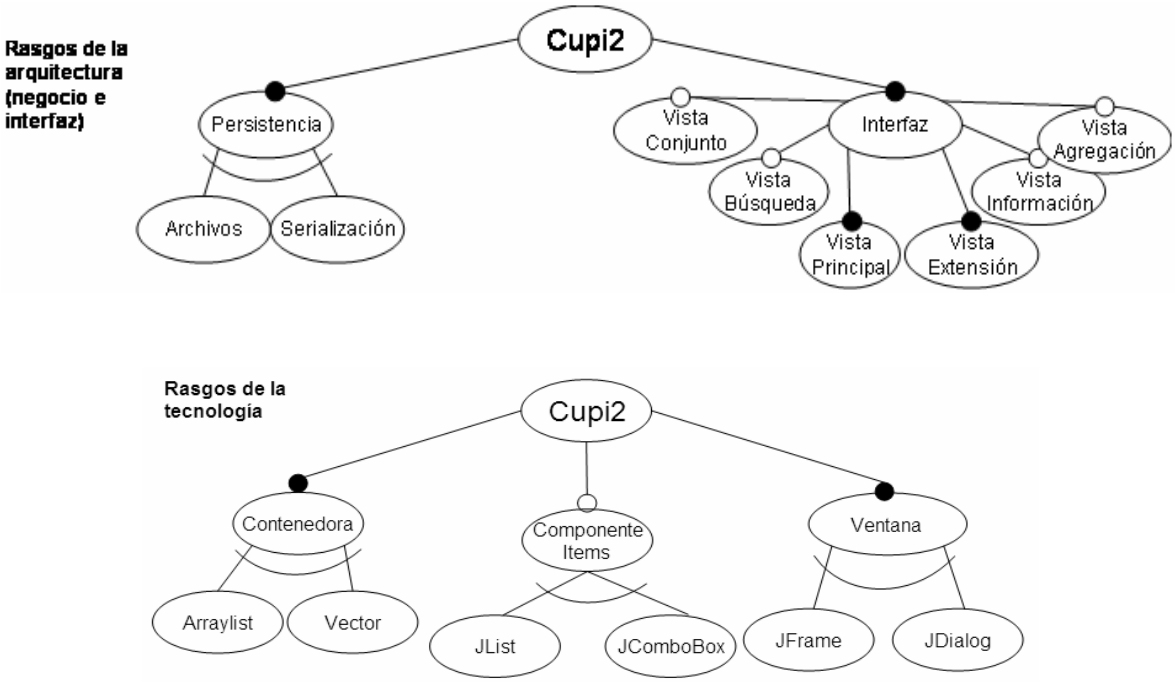

Figura 26. Diagrama de rasgos

Descripción de los modelos de rasgos

Hay dos modelos de rasgos identificados en el dominio de Cupi2:

Rasgos de arquitectura. Contiene rasgos que describen la interfaz gráfica de usuario y los tipos de persistencia disponibles.

Rasgos de tecnología. Contiene rasgos que detallan la implementación en Java.

Cada uno de estos modelos es descrito con mayor detalle en los siguientes párrafos:

Rasgos de la arquitectura

Los rasgos de la arquitectura se dividen en dos grandes categorías:

Interfaz. Describe las formas en que puede visualizarse la información de los elementos de mundo.
Persistencia. Describe las formas en que puede persistirse la información de los elementos del mundo.

Interfaz

Es un rasgo solitario y a la vez un rasgo conjunto. Como rasgo conjunto agrupa los subrasgos vista. Una vista en Cupi2 maneja paneles, listas, listas desplegables, etiquetas, cuadros de texto, botones, imágenes, botones de radio, y cajas de chequeo. Las vistas se implementan con la clase JPanel de Java. Todas las vistas tienen una responsabilidad claramente definida. Los subrasgos vista pueden ser opcionales u obligatorios:

VistaConjunto: es un rasgo opcional. Describe una vista se encarga de visualizar un conjunto de elementos. La vista utiliza componentes como listas o tablas y puede ofrecer adicionalmente servicios que manipulen el orden de presentación de los elementos dentro del conjunto.
VistaBúsqueda: es un rasgo opcional. Describe una vista que se utiliza para realizar búsquedas sobre un conjunto de elementos. Típicamente contiene un cuadro de texto para introducir el índice sobre el cual se debe realizar la búsqueda y un botón de confirmación.
VistaPrincipal: es un rasgo obligatorio. Describe una vista que contiene a las demás vistas y comunica a éstas con el mundo. Todas la interfazs gráficas de usuario tienen una vista principal.
VistaExtensión: es un rasgo obligatorio. Describe una vista que contiene botones y métodos asociados a éstos. En estos métodos sólo se despliega un mensaje de notificación, sin embargo se pueden asociar a ellos nuevas funcionalidades o

extensiones que el estudiante codifique de acuerdo a las instrucciones del profesor. Todas la interfazs gráficas de usuario tienen una vista extensión.
VistaInformación: es un rasgo opcional. Describe una vista que muestra los datos de un elemento del mundo. Además de información alfanumérica, ésta vista también puede mostrar imágenes asociadas a los elementos.
VistaAgregación: es un rasgo opcional. Describe una vista para ingresar información de los elementos del mundo. Sus componentes son similares a los de la vista de información, pero ésta utiliza cuadros de texto y botones de confirmación para recopilar la información ingresada por el usuario de la aplicación.

Persistencia

Es un rasgo solitario opcional y a la vez un rasgo conjunto. Como rasgo conjunto agrupa los rasgos alternativos archivos y serialización.

Archivos: es un rasgo alternativo. Describe un tipo de persistencia sobre archivos planos. Las operaciones de entrada/salida sobre los archivos son codificadas.
Serialización: es un rasgo alternativo. Describe un tipo de persistencia que procesa los elementos de mundo para guardar sus datos en un medio de almacenamiento.

Rasgos de la tecnología

Algunos rasgos de la tecnología están relacionados con los rasgos de la arquitectura. Los que son independientes:

Contenedora
Es un rasgo solitario obligatorio y a la vez un rasgo conjunto. Como conjunto agrupa los rasgos alternativos Arraylist y Vector. En Java, los objetos pueden agruparse usando diferentes tipos de contenedoras, en particular usamos las contenedoras lineales Arraylist y Vector. Tanto Arraylist como Vector describen arreglos de tamaño variable. Un arreglo es un grupo de posiciones de memoria contiguas, las cuales tienen el mismo nombre y el mismo tipo. Las posiciones de memoria almacenan objetos y pueden ser accedidas usando un índice. Tamaño variable significa que el tamaño del arreglo puede crecer o reducirse para soportar la inserción o eliminación de ítems.
Arraylist: no sincronizado, es decir, si múltiples métodos acceden al arraylist concurrentemente, modificando su estructura, puede haber un resultado inesperado. Cuando un objeto es insertado al arraylist, éste incrementa su tamaño en un 50 por ciento.
Vector: sincronizado, es decir, cualquier método que acceda al contenido del vector lo hace de forma segura. Cuando un objeto es insertado al vector, éste incrementa duplica su tamaño.

Los rasgos de la tecnología relacionados con los rasgos de la arquitectura, en particular rasgos de interfaz, son:

Ventana
Es un rasgo conjunto que agrupa los rasgos alternativos JFrame y JDialog. El rasgo Ventana describe tipos de ventanas.
JFrame: es un rasgo alternativo. Describe un tipo de ventana llamado marco. Un marco es una ventana con barra de título y borde. Las aplicaciones Cupi2 tienen un JFrame principal.
JDialog: es un rasgo alternativo. Describe un tipo de ventana llamado cuadro de diálogo. Los cuadros de diálogo se usan comúnmente para obtener información del usuario o para mostrar información al usuario.

El rasgo Ventana está relacionado con los rasgos VistaConjunto, VistaBúsqueda, VistaExtensión, VistaInformación y VistaAgregación, pues todas las vistas deben desplegarse en una ventana. El rasgo VistaPrincipal se relaciona con JFrame de forma obligatoria.

ComponenteItems
Es un rasgo conjunto que contiene los rasgos alternativos JList y JComboBox. ComponenteItems describe un componente gráfico que agrupa una colección de ítems. Tanto JList como JCombobox describen un componente gráfico llamado lista.
JList: es un rasgo alternativo. Describe una lista que exhibe una serie de elementos de los cuales el usuario puede seleccionar uno o más.
JComboBox: es un rasgo alternativo. Describe una lista desplegable y campo editable, el usuario puede seleccionar sólo un valor de la lista.

El rasgo ComponenteItems se relaciona con la VistaConjunto de forma obligatoria, de seleccionarse una VistaConjunto debe especificarse el tipo de ComponenteItems a utilizar.

14. Anexo D. Modelos de entrelazado entre modelos de arquitectura y rasgos de tecnología.

Este anexo presenta entrelazado entre los modelos de arquitectura y los rasgos de tecnología para los ejemplos discotienda y exposición automóviles.

Entrelazados Discotienda

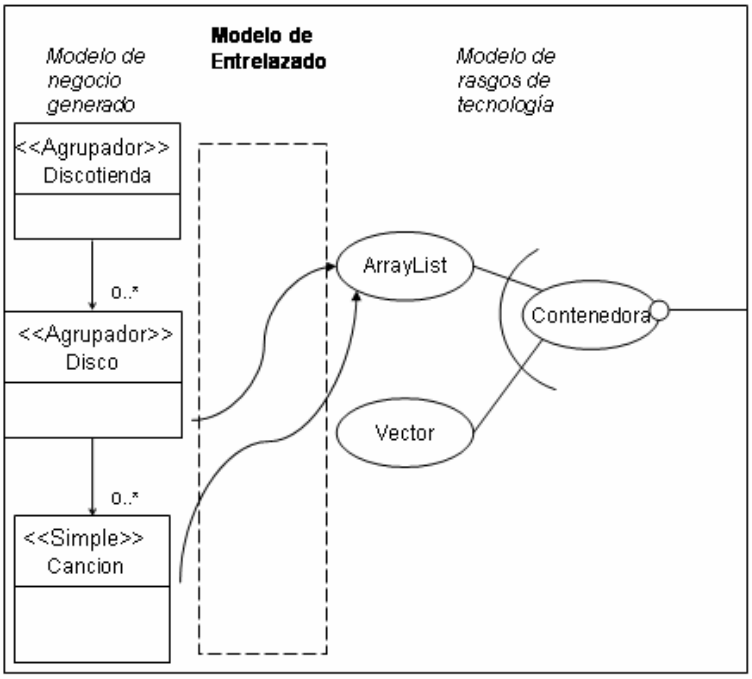

Figura 27. Entrelazado entre el modelo de negocio generado (discotienda) y los rasgos de tecnología.

La figura 27 muestra enlaces que tienen como origen elementos del modelo de negocio de discotienda y como destino rasgos de la tecnología.
El enlace entre el elemento Disco y el rasgo ArrayList y entre éste y Canción indica que las relaciones uno a muchos Discotienda - Disco y Disco – Canción, se representarán con un atributo de tipo ArrayList en Discotienda y Disco respectivamente.

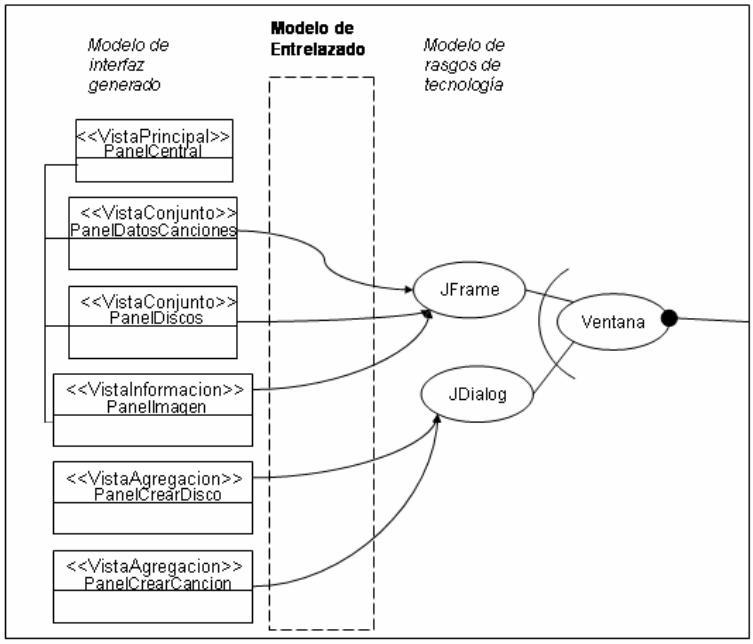

Figura 28. Entrelazado entre el modelo de interfaz generado (discotienda) y los rasgos de tecnología

La figura muestra enlaces que tienen como origen elementos del modelo de interfaz de discotienda y como destino rasgos de la tecnología.

En la figura 28 se hace entrelazado entre los paneles generados y las opciones de Ventana. El enlace entre PanelDatosCanciones y JFrame indica que el panel estará embebido dentro de la ventana principal de la aplicación generada. El enlace entre PanelCrearDisco y JDialog indica que el panel se desplegará en una ventana emergente.

Exposición automóviles

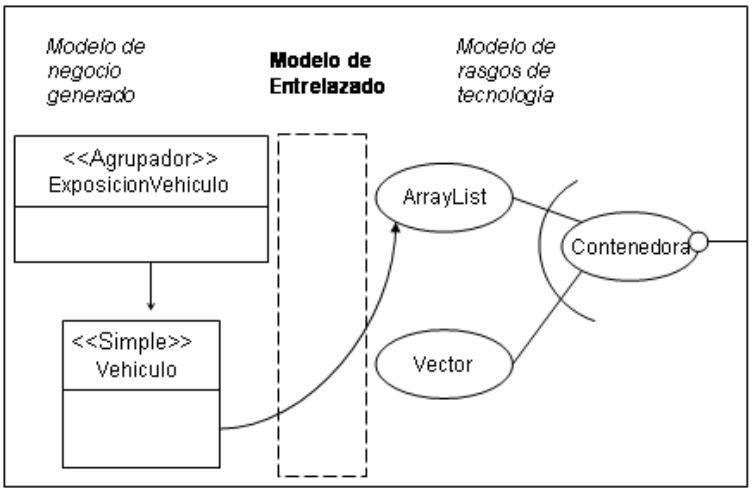

Figura 29. Entrelazado entre el modelo de negocio generado (exposición vehículos) y los rasgos de tecnología

La figura muestra enlaces que tienen como origen elementos del modelo de negocio de exposición vehículos y como destino rasgos de la tecnología.
El enlace entre el elemento Vehículo y el rasgo ArrayList indica que las relaciones uno a muchos ExposiciónVehículo - Vehículo, se representarán con un atributo de tipo ArrayList en ExposiciónVehículo. Note que el ejemplo discotienda también se utilizó este tipo de contenedora.

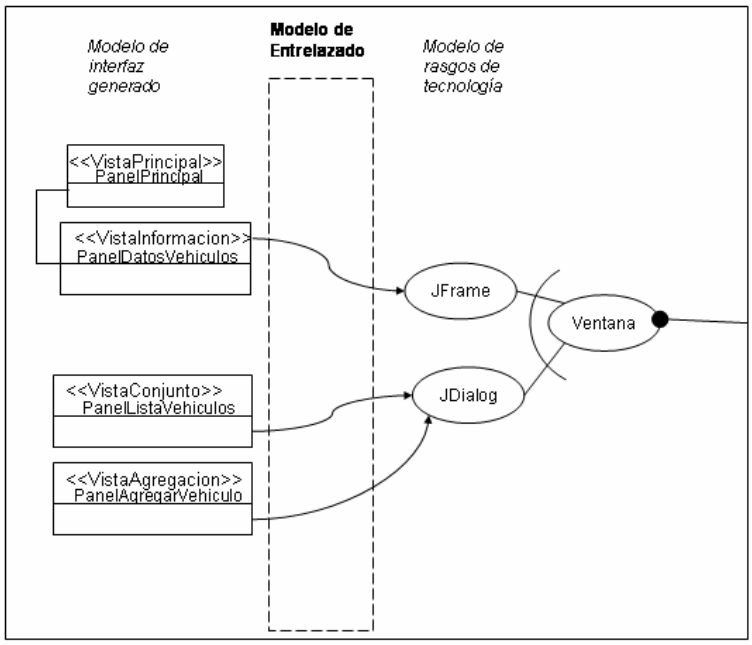

Figura 30. Entrelazado entre el modelo de interfaz generado (exposición vehículos) y los rasgos de tecnología

La figura muestra enlaces que tienen como origen elementos del modelo de interfaz de exposición vehículos y como destino rasgos de la tecnología.

El efecto de los enlaces es similar al descrito en el Modelo de entrelazado entre el modelo de interfaz generado y los rasgos de tecnología del ejemplo discotienda. La diferencia está en los elementos del modelo de interfaz y los tipos de ventana entrelazados.

15. Anexo E. Suposiciones y restricciones sobre el entrelazado de la experimentación.

- Si un elemento de mundo conforme a Agrupador, con el atributo esPrincipal en true, se entrelaza con el rasgo Serialización, todos los elementos contenidos son serializados.

- Todas las vistas tienen un JPanel asociado, el panel se adiciona a un JDialog o al Jframe principal. El panel de VistaPrincipal y VistaExtension siempre se adicionan al Jframe principal.

- Las vistas conjunto, extensión y principal no se muestran en una ventana de tipo JDialog.

16. Anexo F. Manual Qualdev-MDSPL

Qualdev-MDSPL es un plug-in para Eclipse que facilita al usuario la generación de aplicaciones personalizadas. Cada aplicación es miembro de una línea de producto basada en modelos. Qualdev-MDSPL genera código a partir de modelos, metamodelos y transformaciones. Con la generación de código se optimiza el proceso de desarrollo y se logran productos de mayor calidad.

16.1. Requerimientos del plug-in

Qualdev-MDSPL está basado en Eclipse 3.2.1, EMF 2.2.1, ADT 20060630, AMW 20061208. Para generar el código se requiere Acceleo 1.2.

Eclipse está disponible en http://www.eclipse.org/downloads/index.php. Seleccione la versión 3.2.1 (eclipse-SDK-3.2.1-win32.zip).

EMF está disponible en http://www.eclipse.org/emf/downloads/. Seleccione la versión 2.2.1 (emf-sdo-runtime-2.2.1.zip)

ADT puede ser descargado desde http://www.eclipse.org/gmt/atl/download/. Seleccione la versión 20060630 (adt20060630_Eclipse3_1.zip).

AMW puede ser descargado desde http://www.eclipse.org/gmt/amw/download/. Seleccione la versión 20061208 (mwplugins-20061208.zip).

Acceleo puede ser descargado desde http://www.acceleo.org/pages/download-acceleo/en. Seleccione la versión 1.2 (acceleo-1.2.0-runtime.zip).

Siga las instrucciones de instalación de cada uno de los plug-ins citados. Una vez haya finalizado su instalación, descomprima el archivo Qualdev-MDSPL.zip en el directorio \eclipse que ha sido creado en la instalación de Eclipse.

16.2. Dos perspectivas de Qualdev-MDSPL

Qualdev-MDSPL involucra dos roles de usuario: el desarrollador de los activos y el usuario que utiliza el plug-in para generar una aplicación.

16.2.1. Perspectiva del desarrollador

El desarrollador debe crear modelos, metamodelos y transformaciones por cada línea de producto, por lo tanto debe estar familiarizado con las tecnologías utilizadas. Los activos se ubican en el directorio mdspl en la instalación de eclipse. El directorio de los activos se nombra de forma significa, contiene subdirectorios y un archivo de configuración.

Subdirectorios

Los subdirectorios son:
amw. Contiene un metamodelo de entrelazado.
features. Contiene un metamodelo de rasgos y los modelos de rasgos de los modelos destino.
metamodels. Contiene los metamodelos origen y destino.
transfo. Contiene las transformaciones.

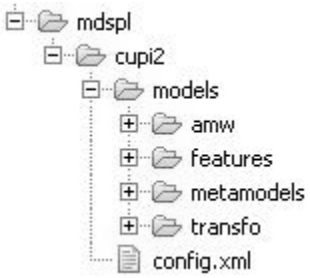

Figura 31. Estructura de directorios de una MD-SPL cupi2

En la distribución del plug-in Qualdev-MDSPL se provee un metamodelo de rasgos y un metamodelo de entrelazado sugeridos en la solución. Los metamodelos adicionales deben ser creados por el desarrollador usando EMF. Para el metamodelo origen debe generar un plug-in EMF que permita la creación de los modelos origen, dicho plug-in debe conectar al eclipse instalado. Las transformaciones son codificadas en el lenguaje ATL, para hacerlo tenga en cuenta los tres tipos de transformaciones de la solución propuesta: base, de control y específicas.

Al utilizar el metamodelo de rasgos los modelos de rasgos pueden ser creados usando los proyectos MMFeature, MMFeature.edit, MMFeature.editor basados en EMF. Importe los proyectos a eclipse, ubíquese en el directorio MMFeature, despliegue el menú contextual, escoja Run->Run as-> Eclipse Application.

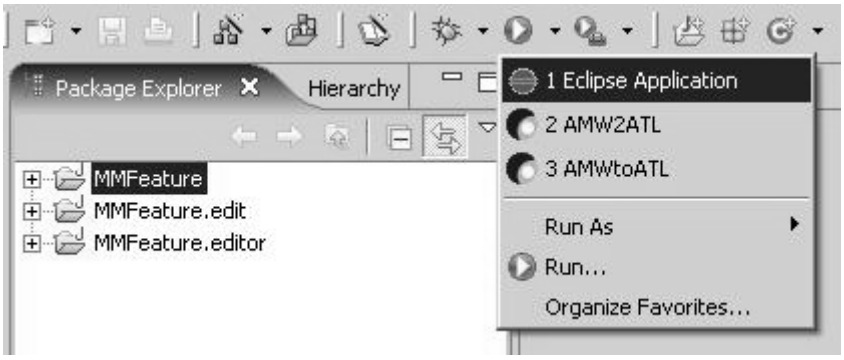

Figura 32. Lanzando una aplicación eclipse.

Una nueva aplicación Eclipse se lanza, allí cree un proyecto simple. Despliegue el menú contextual y escoja New->Other. Seleccione la opción MmFeature dentro de "Example EMF Model Creation Wizard".

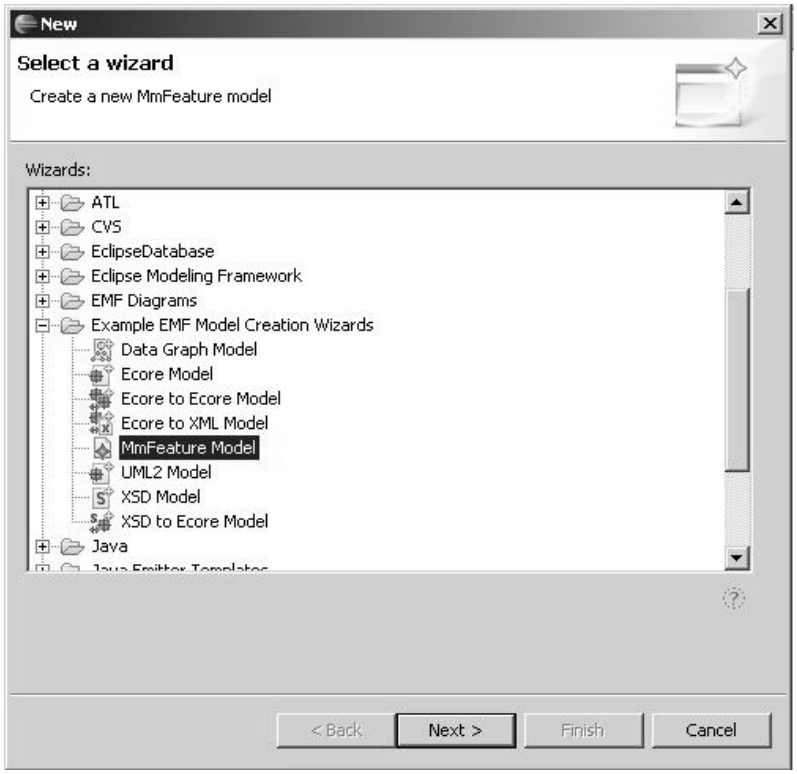

Figura 33. Escogiendo el wizard para construir modelos de rasgos

Especifique un nombre para el modelo de rasgos.

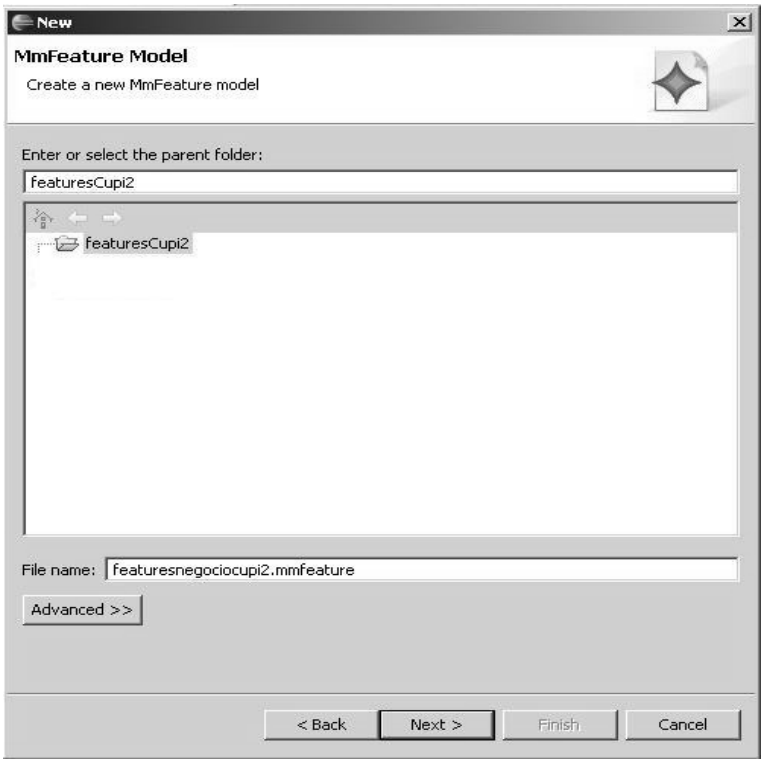

Figura 34. Nombrando el modelo de rasgos

Escoja FeatureModel como la raíz del modelo.

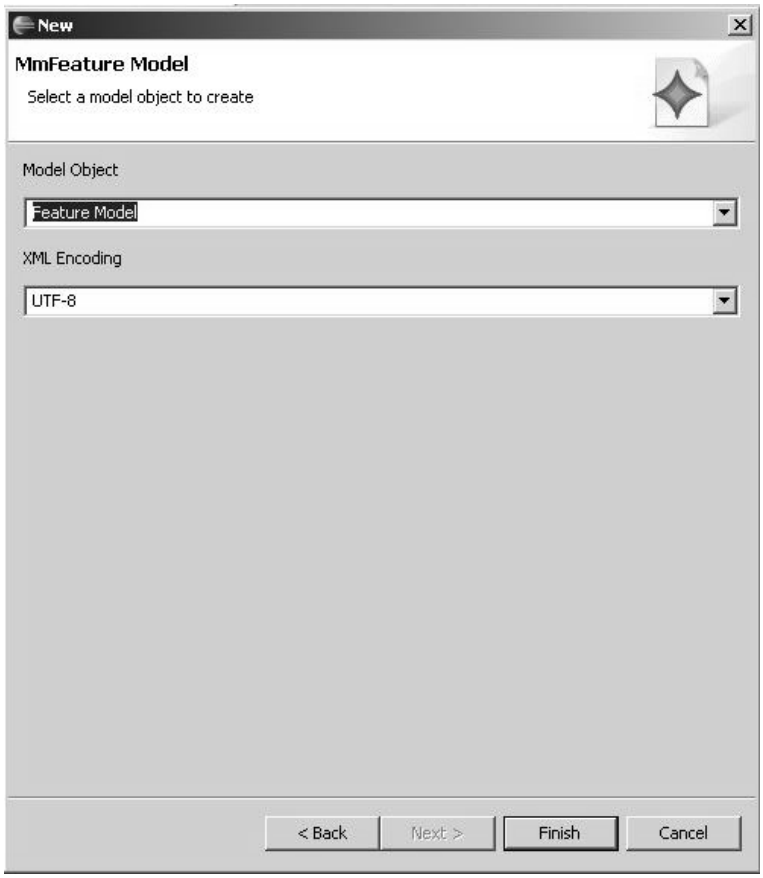

Figura 35. Escogiendo la raíz del modelo de rasgos

Una vista con el modelo de rasgos es desplegado. Desde allí usted podrá adicionar la raíz del modelo de rasgos.

Figura 36. Creando el modelo de rasgos

Para crear un rasgo conjunto seleccione Group. Para crear un rasgo solitario seleccione Cardinality y luego Feature Node. La modificación de los atributos de los elementos se hace desde la vista Properties. Especificar que un rasgo solitario es obligatorio se hace asignando a min y max de Cardinality, 1-1 respectivamente. Especificar que un rasgo solitario es opcional se hace asignando a min y max de Cardinality, 0-1 respectivamente. Especificar que un rasgo conjunto contiene rasgos alternativos se hace asignando a min y max de Group, 1-1 respectivamente.

Archivo de configuración

El archivo de configuración especifica las transformaciones que deben ejecutarse para generar una aplicación. El archivo se nombra “config.xml” y tiene la siguiente estructura:

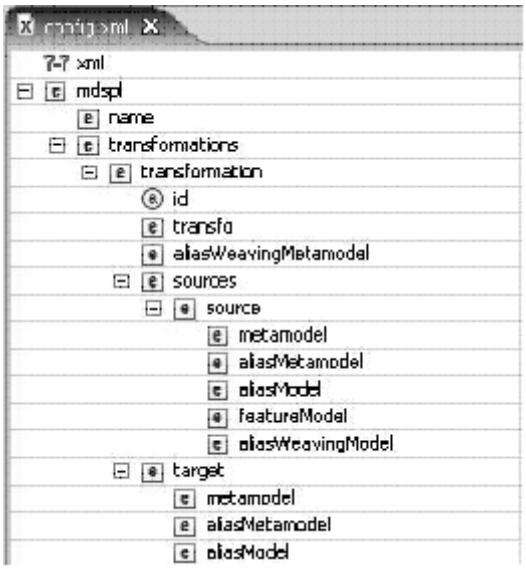

Figura 37. Estructura de config.xml

Definición de tipo de documento (DTD)

<!ELEMENT mdspl (name, transformations)>

name - nombre de la MD-SPL

<!ELEMENT transformations (transformation*)>

<!ELEMENT transformation (transfo, aliasWeavingMetamodel, sources, target)>

<!ATTLIST transformation
id CDATA #REQUIRED
>

id - número que identifica una tranformación
transfo - ubicación del .asm generado por ATL cuando se escribe la transformación

aliasWeavingMetamodel - alias con el que se identifica el metamodelo de entrelazado en las transformaciones

<!ELEMENT sources (source*)>

<!ELEMENT source (metamodel, aliasMetamodel, aliasModel, featureModel, aliasWeavingModel)>

metamodel - ubicación del metamodelo origen
aliasMetamodel - alias con el que se identifica el metamodelo origen en las transformación
aliasModel - alias con el que se identifica el modelo origen en las transformación
featureModel - ubicación del modelo de rasgos
aliasWeavingModel - alias con el que se identifica el modelo de entrelazado en la transformación.

<!ELEMENT target (metamodel, aliasMetamodel, aliasModel)>

metamodel - ubicación del metamodelo destino
aliasMetamodel - alias con el que se identifica el metamodelo destino en la transformación
aliasModel - alias con el que se identifica el modelo destino en la transformación

16.2.2. Perspectiva del usuario

En las siguientes secciones se detallan las funcionalidades proveídas por Qualdev-MDSPL.

El usuario del plug-in debe conocer el esquema de generación de la línea a utilizar. En la siguiente ilustración se referencia el esquema de generación de una MDSPL Cupi2 para el ejemplo discotienda, se sugiere revisar la figura 9 para tener una visión general del esquema de generación.

Seleccionar una MD-SPL

Una vez se despliega la interfaz de eclipse, el primer paso es crear un Proyecto Simple. Lo siguiente es seleccionar la MD-SPL de la aplicación a generar, esto se hace con clic derecho New->Other....

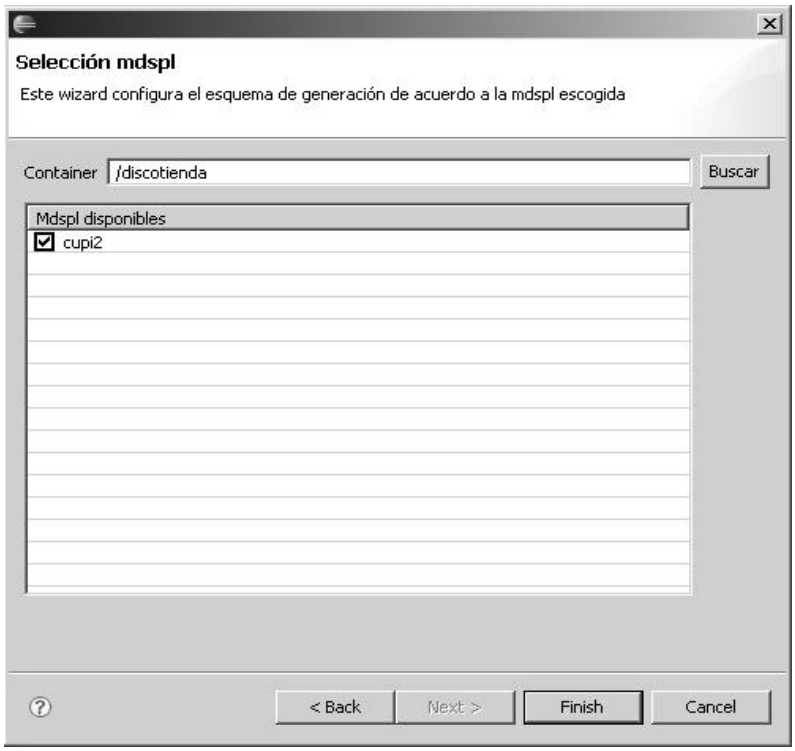

Figura 38. Seleccionando una MD-SPL

Seleccione Mdspl wizard, presione el botón siguiente.

Especifique el proyecto como contenedor usando el botón Buscar.

Selecciona una MD-SPL. Presione el botón Finish

Refresque la vista. Qualdev-MDSPL crea dentro del proyecto el directorio models el cual contiene los activos requeridos durante la generación. Adicionalmente el usuario debe crear el directorio src.

Generación

El usuario debe proveer un modelo origen, realizar operaciones de entrelazado y ejecutar transformaciones. Las operaciones de entrelazado y transformación se habilitan sobre los modelos con extensión xmi. Por organización los modelos generados se ubican en el directorio models\metamodels.

Crear un modelo origen

Con clic derecho sobre el directorio models despliegue el menú contextual. Escoja New->Other... Se despliegan los wizard disponibles, escoja un modelo desde el wizard "Example EMF Model Creation Wizard". Recuerde que el modelo debe ser concerniente a la MD-SPL escogida, para Cupi2 el modelo origen es conforme al metamodelo de mundo.

Especifique un nombre. Especifique el elemento raíz de su modelo. Presione el botón finalizar.

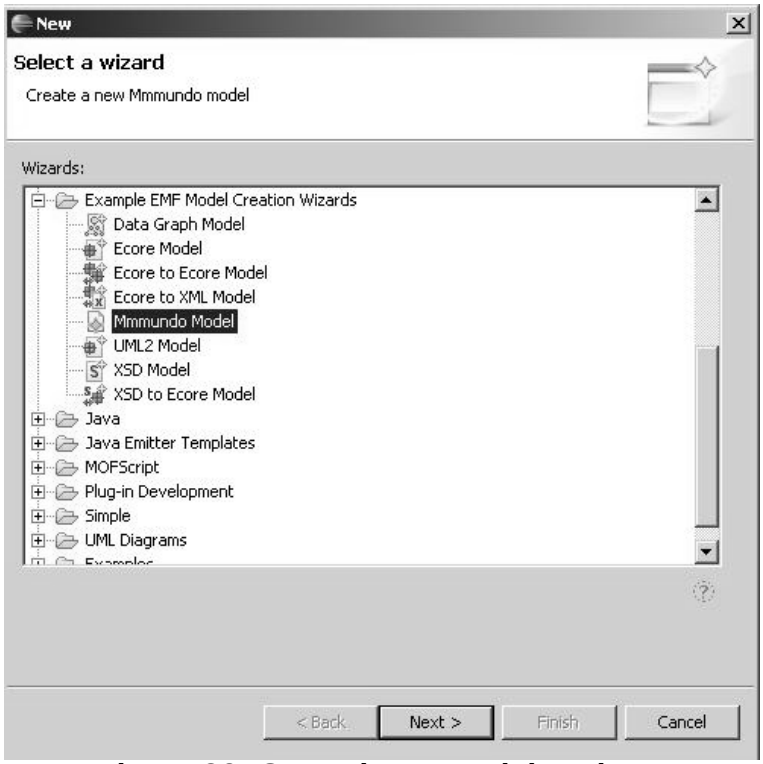

Figura 39. Creando un modelo origen

Cree el modelo adicionando elementos en el árbol. Modifique los atributos de los elementos desde la vista Properties. Cambie la extensión del modelo de mundo por xmi.

Entrelazar

Siguiendo el esquema Cupi2 dos operaciones de entrelazado se efectúan sobre el modelo de mundo creado en el paso anterior. Ubicado en el modelo despliegue el menú emergente con clic derecho. Vaya a la opción Mdspl y luego a Entrelazar.

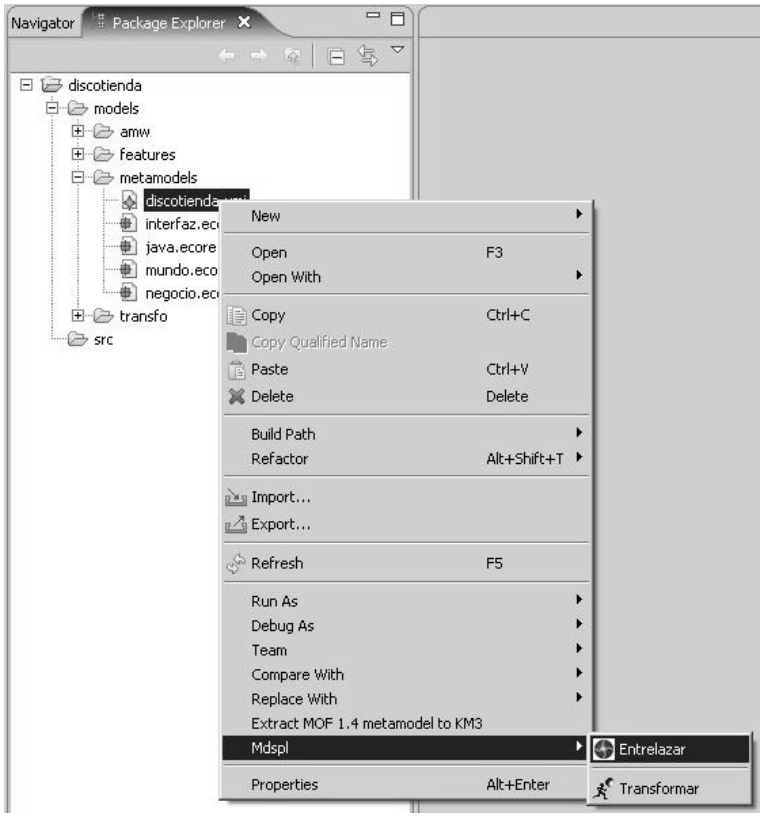

Figura 40. Escogiendo el modelo origen para entrelazar

Qualdev-MDSPL despliega 2 interfazs de AMW. Una de las interfazs contiene en el panel izquierdo el modelo de mundo, en panel central el modelo de entrelazado y en el panel derecho el modelo de rasgos del negocio. La otra contiene en el panel izquierdo el modelo de mundo, en panel central el modelo de entrelazado y en el panel derecho el modelo de rasgos de la interfaz.

Panel Izquierdo
Modelo de mundo

Panel Central
Modelo de entrelazado

Panel Derecho
Modelo de rasgos del negocio

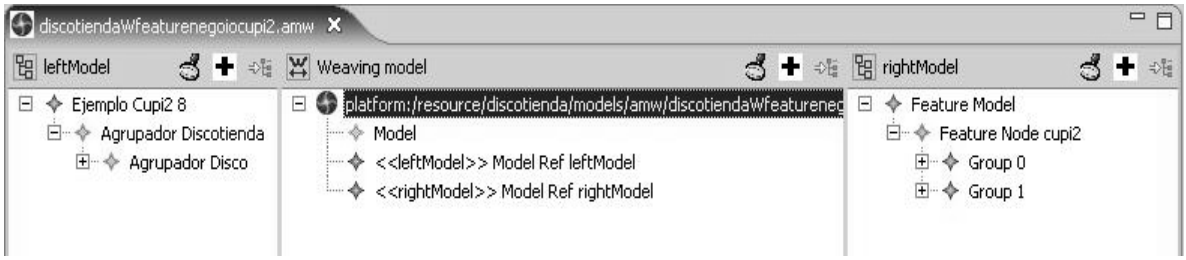

Figura 41. Interfaz de entrelazado

Para entrelazar los elementos del modelo origen y del modelo de rasgos ubíquese en el panel central. Presione clic derecho sobre Model para adicionar Links.

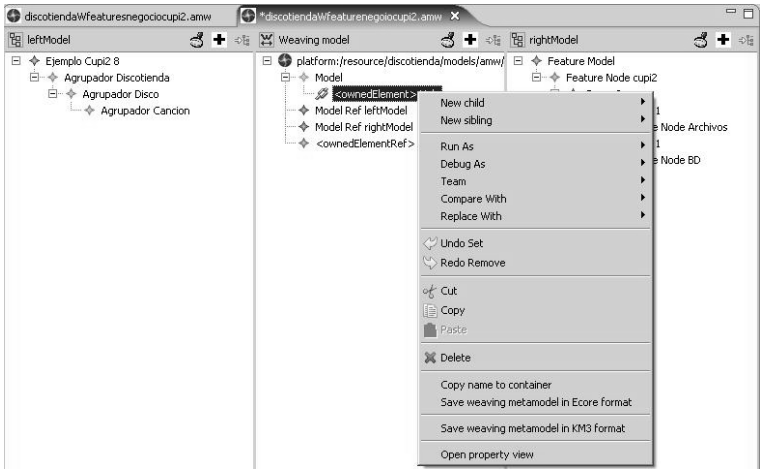

Figura 42. Adicionando enlaces al modelo de rasgos.

Por cada Link adicione un LeftLink y un RightLink.

Arrastre desde el panel izquierdo un elemento del modelo origen y ubíquelo en LeftLink. Repita esta operación desde el panel derecho.

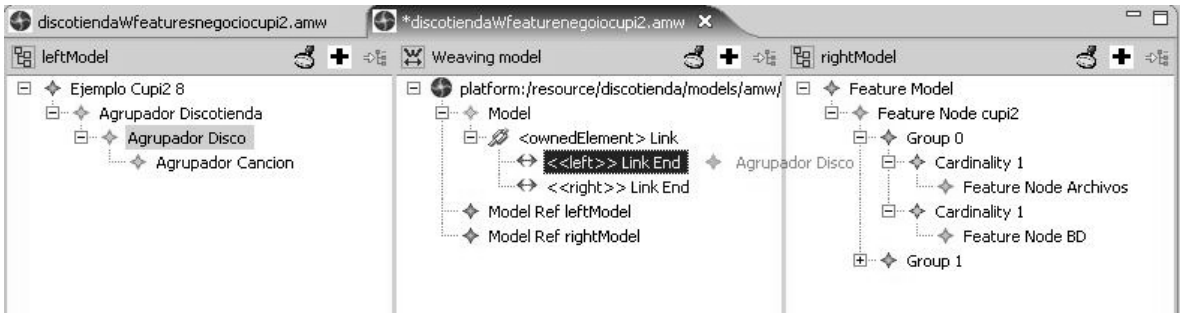

Figura 43. Adicionando derecha e izquierda a cada enlace

Adicione tantos Link como requiera y guarde el modelo de entrelazado desde File->Save.

Recuerde que un mismo elemento del origen no puede entrelazarse con más de un rasgo alternativo. Un mismo elemento origen no debería entrelazarse más de una vez con el mismo rasgo.

Transformar

Cuando se han finalizado las dos operaciones de entrelazado, ubíquese en el modelo origen, despliegue el menú emergente con clic derecho. Vaya a la opción Mdspl y luego a Transformar.

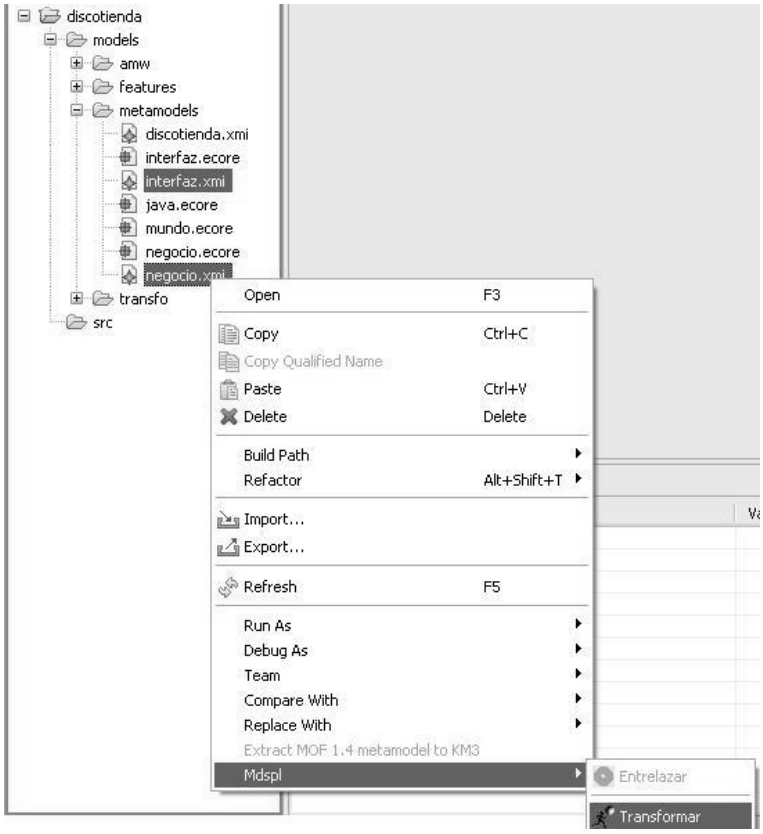

Figura 44. Escogiendo el modelo origen a transformar.

Qualdev-MDSPL ejecuta la transformación adecuada a partir del modelo de mundo y los dos modelos de entrelazados creados en el paso anterior.

Refresque la vista. El directorio models\metamodels debe contener dos modelos destino uno de negocio y otro de interfaz.

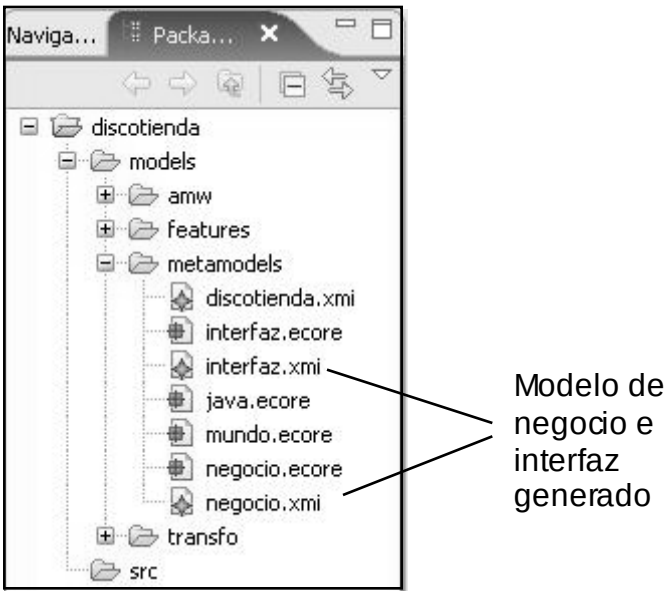

Figura 45. Modelos de arquitectura generados

El siguiente paso en el esquema de generación de Cupi2 es entrelazar el modelo de negocio y el modelo de interfaz con los rasgos del modelo java. Cuando las operaciones de entrelazado se finalicen, se selecciona tanto el modelo de negocio como el de interfaz para ejecutar la transformación. El resultado es un modelo de java.

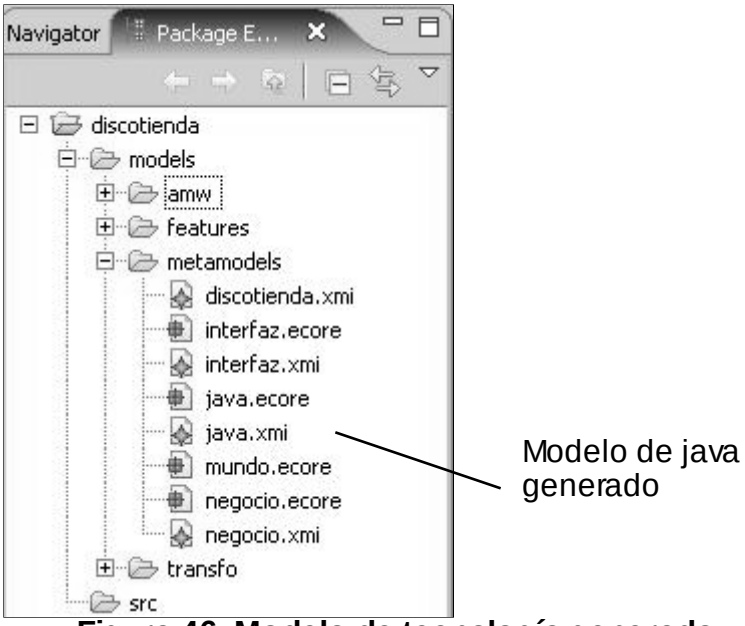

Figura 46. Modelo de tecnología generado

Cuando se ha generado el modelo de java se aplican las plantillas de Acceleo, las cuales generan el código en el directorio src. El código generado se copia al "Source Folder" de un proyecto Java, se agregan clases requeridas que no hayan sido generadas y desde dicho proyecto se ejecuta la aplicación.